CURSO INTENSIVO DE PYTHON

UMA INTRODUÇÃO PRÁTICA E BASEADA EM PROJETOS À PROGRAMAÇÃO

ERIC MATTHES

novatec

CURSO INTENSIVO DE PYTHON

UMA INTRODUÇÃO PRÁTICA E BASEADA EM PROJETOS À PROGRAMAÇÃO

Eric Matthes

Novatec

Para meu pai, que sempre teve tempo de responder às minhas perguntas sobre programação, e para Ever, que está começando a fazer suas perguntas.

SUMÁRIO

4 ■ Trabalhando com listas

12 ■ Uma espaçonave que atira

B ■ Editores de texto

C ■ Obtendo ajuda

Sobre o autor

Eric Matthes é professor de ciências e de matemática no Ensino Médio, mora no Alasca, onde ministra um curso introdutório de Python. Escreve programas desde que tinha cinco anos de idade. Atualmente, Eric tem como foco escrever softwares que visam à falta de eficiência na educação e trazer os benefícios do software de código aberto a essa área. Em seu tempo livre, gosta de escalar montanhas e ficar com sua família.

Sobre o revisor técnico

Kenneth Love é programador Python e professor há vários anos. Deu palestras e tutoriais em conferências, ministrou treinamentos profissionais, foi freelancer de Python e Django e, atualmente, dá aulas em uma empresa de educação online. Kenneth também é coautor do pacote django-braces, que oferece vários mixins práticos para views django baseadas em classe. Você pode acompanhá-lo no Twitter em @*kennethlove*.

AGRADECIMENTOS

Este livro não teria sido possível sem a equipe maravilhosa e extremamente profissional da No Starch Press. Bill Pollock me convidou para escrever um livro introdutório, e sou muito grato por essa proposta original. Tyler Ortman me ajudou a moldar minhas ideias nas primeiras fases das versões preliminares. Os feedbacks iniciais de Liz Chadwick e de Leslie Shen para cada capítulo tiveram valor inestimável, e Anne Marie Walker me ajudou a esclarecer várias partes do livro. Riley Hoffman respondeu a todas as minhas perguntas sobre o processo de compor um livro completo e, com paciência, transformou meu trabalho em um belo produto acabado.

Gostaria de agradecer a Kenneth Love, o revisor técnico do *Curso intensivo de Python*. Conheci Kenneth na PyCon há um ano, e seu entusiasmo pela linguagem e pela comunidade Python tem sido uma fonte constante de inspiração profissional desde então. Kenneth foi além da simples verificação dos fatos e revisou o livro com o objetivo de ajudar programadores iniciantes a desenvolver uma sólida compreensão da linguagem Python e de programação em geral. Apesar do que disse, qualquer falta de precisão que continue existindo é de minha inteira responsabilidade.

Gostaria de agradecer ao meu pai, por ter me introduzido à programação quando eu ainda era bem jovem e por não ter ficado com medo de que eu quebrasse seu equipamento. Gostaria de agradecer à minha esposa, Erin, por ter me apoiado e incentivado durante a escrita deste livro, e gostaria de agradecer ao meu filho Ever, cuja curiosidade me inspira todos os dias.

INTRODUÇÃO

Todo programador tem uma história sobre como aprendeu a escrever seu primeiro programa. Comecei a aprender ainda criança, quando meu pai trabalhava para a Digital Equipment Corporation, uma das empresas pioneiras da era moderna da computação. Escrevi meu primeiro programa em um kit de computador que meu pai havia montado em nosso porão. O computador não tinha nada além de uma placa-mãe básica conectada a um teclado, sem gabinete, e o monitor era apenas um tubo de raios catódicos. Meu primeiro programa foi um jogo simples de adivinhação de números, que tinha um aspecto semelhante a:

```
I'm thinking of a number! Try to guess the number I'm thinking of: 25
Too low! Guess again: 50
Too high! Guess again: 42
That's it! Would you like to play again? (yes/no) no
Thanks for playing!
```

Sempre vou me lembrar de como eu ficava satisfeito ao ver minha família brincar com um jogo que eu havia criado e que funcionava conforme havia planejado.

Essa experiência inicial teve um impacto duradouro. Há uma verdadeira satisfação em criar algo com um propósito, algo que resolva um problema. O software que escrevo hoje em dia atende a uma necessidade mais significativa se comparado aos meus esforços da infância, mas o senso de satisfação proveniente da criação de um programa que funciona ainda é, de modo geral, o mesmo.

A quem este livro se destina?

O objetivo deste livro é deixá-lo pronto para usar Python o mais rápido possível, de modo que você possa criar programas que funcionem – jogos, visualizações de dados e aplicações web – ao mesmo tempo que desenvolve uma base em programação que terá muita utilidade pelo resto de sua vida. *Curso intensivo de Python* foi escrito para pessoas de qualquer idade, que jamais programaram em Python antes ou que nunca programaram. Se quiser conhecer o básico sobre programação rapidamente para poder se concentrar em projetos interessantes e quiser testar sua compreensão acerca de novos conceitos resolvendo problemas significativos, este livro é ideal para você. *Curso intensivo de Python* também é perfeito para professores de Ensino Fundamental e Médio que queiram oferecer uma introdução à programação baseada em projetos aos seus alunos.

O que você pode esperar do livro?

O propósito deste livro é fazer de você um bom programador, em geral, e um bom programador de Python, em particular. Você aprenderá de modo eficiente e adotará bons hábitos à medida que eu lhe proporcionar uma base sólida em conceitos gerais de programação. Após trabalhar com o *Curso Intensivo de Python*, você deverá estar pronto para passar para técnicas mais avançadas de Python, e será mais fácil ainda dominar sua próxima linguagem de programação.

Na primeira parte deste livro, você conhecerá os conceitos básicos de programação necessários para saber escrever programas Python. Esses conceitos são os mesmos que você aprenderia ao começar com praticamente qualquer linguagem de programação. Você conhecerá os diferentes tipos de dados e as maneiras de armazená-los em listas e em dicionários em seus programas. Aprenderá a criar coleções de dados e a trabalhar com essas coleções de modo eficiente. Conhecerá os laços `while` e `if` para testar determinadas condições a fim de poder executar seções específicas de código enquanto essas condições forem verdadeiras e executar outras seções quando não forem – uma técnica que ajuda bastante a automatizar processos.

Você aprenderá a aceitar entradas de usuários para deixar seus programas

interativos e mantê-los executando enquanto o usuário estiver ativo. Explorará o modo de escrever funções para deixar partes de seu programa reutilizáveis; dessa forma, será necessário escrever blocos de código que executem determinadas ações apenas uma vez e poderá usá-los quantas vezes quiser. Então você estenderá esse conceito para comportamentos mais complicados usando classes, criando programas bem simples que respondam a uma variedade de situações. Além disso, aprenderá a escrever programas que tratam erros comuns com elegância. Depois de trabalhar com cada um desses conceitos básicos, você escreverá alguns programas pequenos que resolverão problemas bem definidos. Por fim, você dará seu primeiro passo em direção à programação de nível intermediário, aprendendo a escrever testes para seu código de modo a poder desenvolver mais os seus programas sem se preocupar com a introdução de bugs. Todas as informações da Parte I prepararão você para assumir projetos maiores e mais complexos.

Na Parte II, você aplicará o que aprendeu na Parte I em três projetos. Você pode desenvolver todos os projetos, ou qualquer um deles, na ordem que lhe for mais apropriada. No primeiro projeto (Capítulos 12 a 14), você criará um jogo de tiros chamado Alien Invasion, no estilo do Space Invaders, constituído de níveis com dificuldade crescente. Após concluir esse projeto, você deverá estar bem encaminhado para desenvolver seus próprios projetos 2D.

O segundo projeto (Capítulos 15 a 17) apresenta a visualização de dados a você. Os cientistas de dados procuram compreender a enorme quantidade de informações disponíveis a eles por meio de várias técnicas de visualização. Você trabalhará com conjuntos de dados gerados por meio de código, conjuntos de dados baixados de fontes online e conjuntos de dados baixados automaticamente pelos seus programas. Depois de concluir esse projeto, você será capaz de escrever programas que filtrem conjuntos grandes de dados e criem representações visuais dessas informações armazenadas.

No terceiro projeto (Capítulos 18 a 20), você criará uma pequena aplicação web chamada Learning Log. Esse projeto permite manter um diário com ideias e conceitos que você aprendeu sobre um assunto

específico. Você será capaz de manter registros separados para assuntos diferentes e permitirá que outras pessoas criem uma conta e comecem a escrever seus próprios diários. Aprenderá também a implantar seu projeto para que qualquer pessoa possa acessá-lo online de qualquer lugar.

Por que Python?

Todos os anos eu avalio se devo continuar usando Python ou se devo mudar para uma linguagem diferente – talvez uma que seja mais nova no mundo da programação. Porém, continuo focando em Python por diversos motivos. Python é uma linguagem extremamente eficiente: seus programas farão mais com menos linhas de código, se comparados ao que muitas outras linguagens exigiriam. A sintaxe de Python também ajudará você a escrever um código "limpo". Seu código será fácil de ler, fácil de depurar, fácil de estender e de expandir, quando comparado com outras linguagens.

As pessoas usam Python para muitos propósitos: criar jogos, construir aplicações web, resolver problemas de negócios e desenvolver ferramentas internas em todo tipo de empresas interessantes. Python também é intensamente usada em áreas científicas para pesquisa acadêmica e trabalhos aplicados.

Um dos principais motivos pelos quais continuo a usar Python é por causa de sua comunidade, que inclui um grupo de pessoas incrivelmente diversificado e acolhedor. A comunidade é essencial aos programadores, pois a programação não é um objetivo a ser perseguido de forma solitária. A maioria de nós, mesmo os programadores mais experientes, precisa pedir conselhos a outras pessoas que já tenham resolvido problemas semelhantes. Ter uma comunidade bem conectada, que ofereça bastante suporte, é fundamental para ajudar você a resolver problemas, e a comunidade Python apoia totalmente pessoas como você, que estão aprendendo Python como a primeira linguagem de programação.

Python é uma ótima linguagem para aprender, portanto, vamos começar!

PARTE I

CONCEITOS BÁSICOS

A Parte I deste livro ensina os conceitos básicos de que você precisará para escrever programas Python. Muitos desses conceitos são comuns a todas as linguagens de programação, portanto, serão úteis ao longo de sua vida como programador.

No **Capítulo 1** você instalará Python em seu computador e executará seu primeiro programa, que exibe a mensagem *Hello world!* na tela.

No **Capítulo 2** você aprenderá a armazenar informações em variáveis e a trabalhar com texto e valores numéricos.

Os **Capítulos 3 e 4** apresentam as listas. As listas podem armazenar tantas informações quantas você quiser em uma variável, permitindo trabalhar com esses dados de modo eficiente. Você será capaz de trabalhar com centenas, milhares e até mesmo milhões de valores com apenas algumas linhas de código.

No **Capítulo 5** veremos o uso de instruções if para escrever código que responda de uma maneira se determinadas condições forem verdadeiras, e respondam de outra se essas condições forem falsas.

O **Capítulo 6** mostra como usar dicionários em Python, que permitem fazer conexões entre informações diferentes. Assim como as listas, os dicionários podem conter tantas informações quantas forem necessárias armazenar.

No **Capítulo 7** você aprenderá a aceitar entradas de usuário para deixar seus programas interativos. Conhecerá também os laços `while`, que

executam blocos de código repetidamente enquanto determinadas condições permanecerem verdadeiras.

No **Capítulo 8** você escreverá funções: são blocos de código nomeados que executam uma tarefa específica e podem ser executados sempre que forem necessários.

O **Capítulo 9** apresenta as classes, que permitem modelar objetos do mundo real, como cachorros, gatos, pessoas, carros, foguetes e muito mais, de modo que seu código possa representar qualquer entidade real ou abstrata.

O **Capítulo 10** mostra como trabalhar com arquivos e tratar erros para que seus programas não falhem de forma inesperada. Você armazenará dados antes que seu programa termine e os lerá de volta quando o programa executar novamente. Conhecerá as exceções de Python, que permitem antecipar erros, e fará seus programas tratarem esses erros de modo elegante.

No **Capítulo 11** você aprenderá a escrever testes para seu código a fim de verificar se seus programas funcionam conforme esperado. Como resultado, será capaz de expandir seus programas sem se preocupar com a introdução de novos bugs. Testar seu código é uma das primeiras habilidades que ajudarão você a fazer a transição de um programador iniciante para um programador intermediário.

1

INICIANDO

Neste capítulo você executará seu primeiro programa Python, *hello_world.py*. Em primeiro lugar, verifique se Python está instalado em seu computador; se não estiver, instale-o. Instale também um editor de texto para trabalhar com seus programas Python. Editores de texto reconhecem código Python e enfatizam determinadas seções à medida que você escrever o código, facilitando entender a sua estrutura.

Configurando seu ambiente de programação

Python difere um pouco em sistemas operacionais distintos, portanto, você deverá ter algumas considerações em mente. Veremos aqui duas versões importantes de Python atualmente em uso, e apresentaremos os passos para configurar Python em seu sistema.

Python 2 e Python 3

Atualmente, duas versões de Python estão disponíveis: Python 2 e Python 3, a mais recente. Toda linguagem de programação evolui à medida que surgem novas ideias e tecnologias, e os desenvolvedores de Python têm deixado a linguagem mais versátil e eficaz de forma contínua. A maioria das alterações é incremental e dificilmente perceptível, mas, em alguns casos, um código escrito para Python 2 poderá não executar de modo apropriado em sistemas com a versão Python 3 instalada. Ao longo deste livro, destacarei as áreas em que houver diferenças significativas entre Python 2 e Python 3, portanto, independentemente da versão usada, você poderá seguir as instruções.

Se as duas versões estiverem instaladas em seu sistema ou se precisar instalar Python, instale Python 3. Se Python 2 for a única versão em seu sistema e você prefira escrever código de imediato em vez de instalar

Python, poderá começar com Python 2. Porém, quanto antes fizer o upgrade para Python 3, melhor será, pois você estará trabalhando com a versão mais recente.

Executando trechos de código Python

Python tem um interpretador que executa em uma janela de terminal, permitindo que você teste porções de código Python sem precisar salvar e executar um programa completo.

Neste livro você verá trechos de código semelhantes a:

```
u >>> print("Hello Python interpreter!")
Hello Python interpreter!
```

O texto em negrito é o que você digitará e então executará teclando ENTER. A maioria dos exemplos do livro são programas pequenos, autocontidos, que serão executados a partir de seu editor, pois é assim que você escreverá a maior parte de seu código. Às vezes, porém, conceitos básicos serão mostrados em uma série de trechos de código executados em uma sessão de terminal Python para demonstrar conceitos isolados de modo mais eficiente. Sempre que você vir os três sinais de maior em uma listagem de código u, estará vendo a saída de uma sessão de terminal. Experimentaremos programar no interpretador de seu sistema em breve.

Hello World!

Uma crença de longa data no mundo da programação é que exibir uma mensagem *Hello world!* na tela como seu primeiro programa em uma nova linguagem trará sorte.

Em Python, podemos escrever o programa *Hello World* com uma linha:

```
print("Hello world!")
```

Um programa simples como esse tem um verdadeiro propósito. Se executar corretamente em seu sistema, qualquer programa Python que você escrever também deverá funcionar. Em breve, veremos como escrever esse programa em seu sistema em particular.

Python em sistemas operacionais diferentes

Python é uma linguagem de programação para diversas plataformas, o que significa que ela executará em todos os principais sistemas operacionais. Qualquer programa Python que você escrever deverá executar em qualquer computador moderno que tenha Python instalado. No entanto, os métodos para configurar Python em diferentes sistemas operacionais variam um pouco.

Nesta seção aprenderemos a configurar Python e a executar o programa *Hello World* em seu próprio sistema. Em primeiro lugar, verifique se Python está instalado em seu sistema e instale-o, se não estiver. Em seguida, você instalará um editor de texto simples e salvará um arquivo Python vazio chamado *hello_world.py*. Por fim, executará o programa *Hello World* e solucionará qualquer problema que houver. Acompanharei você nesse processo em cada sistema operacional para que você tenha um ambiente de programação Python amigável a iniciantes.

Python no Linux

Sistemas Linux são projetados para programação, portanto, Python já está instalado na maioria dos computadores Linux. As pessoas que escrevem e dão manutenção em Linux esperam que você programe por conta própria em algum momento e o incentivam a fazê-lo. Por esse motivo, há poucos itens que precisam ser instalados e poucos parâmetros que devem ser alterados para começar a programar.

Verificando sua versão de Python

Abra uma janela de terminal executando a aplicação Terminal em seu sistema (no Ubuntu, você pode pressionar CTRL-ALT-T). Para descobrir se Python está instalado, digite **`python`** com *p* minúsculo. Você deverá ver uma saída que informa qual versão de Python está instalada e um prompt `>>>` em que poderá começar a fornecer comandos Python, assim:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Essa saída informa que, no momento, Python 2.7.6 é a versão-padrão de Python instalada nesse computador. Quando vir essa saída, pressione CTRL-D ou digite `exit()` para sair do prompt de Python e retornar a um prompt do terminal.

Para verificar se Python 3 está instalada, talvez você precise especificar essa versão; portanto, mesmo que a saída tenha mostrado Python 2.7 como a versão-padrão, experimente executar o comando `python3`:

```
$ python3
Python 3.5.0 (default, Sep 17 2015, 13:05:18)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Essa saída quer dizer que você também tem Python 3 instalada, portanto, poderá usar qualquer uma das versões. Sempre que vir o comando `python` neste livro, digite `python3` em seu lugar. A maioria das distribuições Linux já tem Python instalada, mas se, por algum motivo, a sua não tiver ou se seu sistema veio com Python 2 e você quer instalar Python 3, consulte o Apêndice A.

Instalando um editor de texto

O Geany é um editor de texto simples: é fácil de instalar, permitirá executar quase todos os seus programas diretamente do editor em vez do terminal, dá destaque à sintaxe para colorir seu código e executa-o em uma janela do terminal; desse modo, você se acostumará a usar terminais. O Apêndice B tem informações sobre outros editores de texto, porém recomendo usar o Geany, a menos que você tenha um bom motivo para utilizar um editor diferente.

Você pode instalar o Geany com uma linha na maioria dos sistemas Linux:

```
$ sudo apt-get install geany
```

Se isso não funcionar, veja as instruções em

http://geany.org/Download/ThirdPartyPackages/.

Executando o programa Hello World

Para começar seu primeiro programa, abra o Geany. Pressione a tecla Super (muitas vezes chamada de tecla Windows) e procure o Geany em seu sistema. Crie um atalho arrastando o ícone para sua barra de tarefas ou o desktop. Em seguida, crie uma pasta para seus projetos em algum lugar de seu sistema e chame-a de *python_work*. (É melhor usar letras minúsculas e underscores para espaços em nomes de arquivos e de pastas, pois fazem parte das convenções de nomenclatura de Python.) Volte para o Geany e salve um arquivo Python vazio (**File ▸ Save As**, ou Arquivo ▸ Salvar como) chamado *hello_world.py* em sua pasta *python_work*. A extensão *.py* informa o Geany que seu arquivo conterá um programa Python. Também diz ao Geany como executar seu programa e dar destaque ao texto de modo conveniente.

Após ter salvado seu arquivo, forneça a linha a seguir:

```
print("Hello Python world!")
```

Se várias versões de Python estiverem instaladas em seu sistema, será preciso garantir que o Geany esteja configurado para utilizar a versão correta. Acesse **Build ▸ Set Build Commands** (Construir ▸ Definir Comandos de Construção). Você deverá ver as palavras *Compile* e *Execute* com um comando ao lado de cada um. O Geany pressupõe que o comando correto para cada um é `python`, mas se seu sistema utiliza o comando `python3`, será necessário alterar essa informação.

Se o comando `python3` funcionou em uma sessão de terminal, mude os comandos Compile e Execute para que o Geany utilize o interpretador do Python 3. Seu comando Compile deve ser:

```
python3 -m py_compile "%f"
```

É preciso digitar esse comando exatamente como mostrado. Certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas estejam exatamente iguais ao que foi mostrado aqui.

Seu comando Execute deve ser:

```
python3 "%f"
```

Novamente, certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas estejam iguais ao que foi mostrado aqui. A Figura 1.1 mostra como devem ser esses comandos no menu de configuração do Geany.

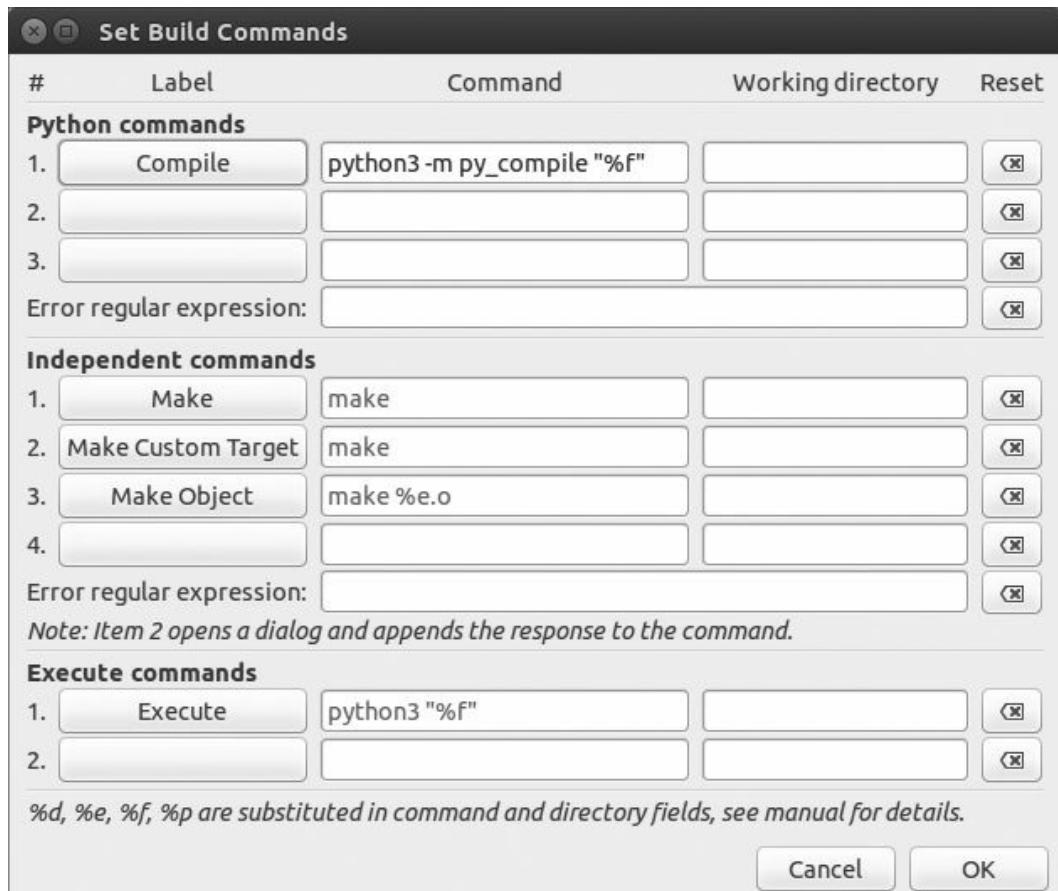



Figura 1.1 – Nesse caso, o Geany está configurado para usar Python 3 no Linux.

Agora execute *hello_world.py* selecionando **Build ▸ Execute** (Construir ▸ Executar) no menu, clicando no ícone Execute (Executar, que mostra um conjunto de engrenagens) ou pressionando F5. Uma janela de terminal deve aparecer com a saída a seguir:

```
Hello Python world!

------------------
(program exited with code: 0)
Press return to continue
```

Se não vir isso, confira todos os caracteres da linha que você inseriu. Você não usou letra maiúscula acidentalmente em `print`? Você não esqueceu uma das aspas, ou as duas, ou os parênteses? As linguagens de programação esperam uma sintaxe bem específica e, se você não a fornecer, haverá erros. Se não conseguir fazer o programa executar, veja a seção "Resolvendo problemas de instalação".

Executando Python em uma sessão de terminal

Você pode tentar executar trechos de código Python abrindo um terminal e digitando `python` ou `python3`, como fizemos quando conferimos sua versão. Faça isso novamente, porém, desta vez, digite a linha a seguir na sessão do terminal:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Você deverá ver sua mensagem exibida diretamente na janela de terminal usada no momento. Lembre-se de que você pode fechar o interpretador Python teclando CTRL-D ou digitando o comando `exit()`.

Python no OS X

Python já vem instalado na maioria dos sistemas OS X. Depois que tiver certeza de que Python está presente, você precisará instalar um editor de texto e garantir que esteja configurado corretamente.

Verificando se Python está instalado

Abra uma janela de terminal acessando **Applications ▸ Utilities ▸ Terminal** (Aplicativos ▸ Utilitários ▸ Terminal). Você também pode pressionar COMMAND-barra de espaço, digitar **`terminal`** e, então, teclar ENTER. Para descobrir se Python está instalado, digite **`python`** com *p* minúsculo. Você deverá ver uma saída que informa qual versão de Python está instalada em seu sistema e um prompt `>>>` em que poderá começar a fornecer comandos Python, assim:

```
$ python
```

```
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC  4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

Essa saída informa que, no momento, Python 2.7.5 é a versão-padrão de Python instalada nesse computador. Quando vir essa saída, pressione CTRL-D ou digite `exit()` para sair do prompt de Python e retornar a um prompt do terminal.

Para conferir se você tem Python 3, experimente executar o comando `python3`. Você pode obter uma mensagem de erro, mas se a saída mostrar que Python 3 está instalada, você poderá usar essa versão sem precisar instalá-la. Se `python3` funcionar em seu sistema, sempre que vir o comando `python` neste livro, lembre-se de usar `python3` em seu lugar. Se, por algum motivo, seu sistema não veio com Python ou se você só tem Python 2 e quer instalar Python 3 agora, consulte o Apêndice A.

Executando Python em uma sessão de terminal

Você pode testar trechos de código Python abrindo um terminal e digitando `python` ou `python3`, como fizemos quando conferimos nossa versão. Faça isso novamente, porém, dessa vez, digite a linha a seguir na sessão do terminal:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Você deverá ver sua mensagem exibida diretamente na janela de terminal usada no momento. Lembre-se de que o interpretador Python pode ser fechado teclando CTRL-D ou digitando o comando `exit()`.

Instalando um editor de texto

O Sublime Text é um editor de texto simples: é fácil de instalar no OS X, permitirá executar quase todos os seus programas diretamente do editor em vez do terminal, dá destaque à sintaxe para colorir seu código e executa-o em uma sessão de terminal incluída na janela do Sublime Text para que seja mais fácil ver a saída. O Apêndice B tem informações sobre

outros editores de texto, mas recomendo usar o Sublime Text, a menos que você tenha um bom motivo para utilizar um editor diferente.

Você pode fazer download do instalador do Sublime Text a partir de *http://sublimetext.com/3*. Clique no link para download e procure um instalador para OS X. O Sublime Text tem uma política de licença bem liberal: você pode usar o editor gratuitamente durante o tempo que quiser, mas o autor pede que você compre uma licença se gostar dele e quiser utilizá-lo continuamente. Após ter baixado o instalador, abra-o e então arraste o ícone do Sublime Text para sua pasta *Applications*.

Configurando o Sublime Text para Python 3

Se você usa um comando diferente de `python` para iniciar uma sessão de terminal Python, será necessário configurar o Sublime Text para que ele saiba onde encontrar a versão correta de Python em seu sistema. Dê o seguinte comando para descobrir o path completo de seu interpretador Python:

```
$ type -a python3
python3 is /usr/local/bin/python3
```

Agora abra o Sublime Text e acesse **Tools ▸ Build System ▸ New Build System** (Ferramentas ▸ Sistema de Construção ▸ Novo Sistema de Construção), que abrirá um novo arquivo de configuração para você. Apague o que vir e insira o seguinte:

Python3.sublime-build

```
{
    "cmd": ["/usr/local/bin/python3", "-u", "$file"],
}
```

Esse código diz ao Sublime Text para usar o comando `python3` de seu sistema quando executar o arquivo aberto no momento. Lembre-se de usar o path encontrado ao executar o comando `type -a python3` no passo anterior. Salve o arquivo como *Python3.sublime-build* no diretório default aberto pelo Sublime Text quando escolher Save (Salvar).

Executando o programa Hello World

Para começar seu primeiro programa, inicie o Sublime Text abrindo a pasta *Applications* e dando um clique duplo no ícone desse editor. Você também pode pressionar COMMAND-barra de espaço e inserir **`sublime text`** na barra de pesquisa que aparecer.

Crie uma pasta chamada *python_work* para seus projetos em algum lugar de seu sistema. (É melhor usar letras minúsculas e underscores para espaços em nomes de arquivos e de pastas, pois fazem parte das convenções de nomenclatura de Python.) Salve um arquivo Python vazio (**File ▸ Save As**, ou Arquivo ▸ Salvar como) chamado *hello_world.py* em sua pasta *python_work*. A extensão *.py* informa o Sublime Text que seu arquivo conterá um programa Python e lhe diz como executar seu programa e dar destaque ao texto de forma conveniente.

Após ter salvado seu arquivo, insira a linha a seguir:

```
print("Hello Python world!")
```

Se o comando `python` funciona em seu sistema, você poderá executar seu programa selecionando **Tools ▸ Build** (Ferramentas ▸ Construir) no menu ou pressionando CTRL-B. Se você configurou o Sublime Text para usar um comando diferente de `python`, selecione **Tools ▸ Build System** (Ferramentas ▸ Sistema de Construção) e então selecione **Python 3**. Isso define Python 3 como a versão default de Python, e você poderá selecionar **Tools ▸ Build** ou simplesmente pressionar COMMAND-B para executar seus programas a partir de agora.

Uma tela do terminal deve aparecer na parte inferior da janela do Sublime Text, mostrando a saída a seguir:

```
Hello Python world!
[Finished in 0.1s]
```

Se não vir isso, confira todos os caracteres da linha que você inseriu. Você não usou uma letra maiúscula acidentalmente em `print`? Você não esqueceu uma das aspas, ou as duas, ou os parênteses? As linguagens de programação esperam uma sintaxe bem específica e, se você não a fornecer, haverá erros. Se não conseguir fazer o programa executar, veja a seção “Resolvendo problemas de instalação”.

Python no Windows

O Windows nem sempre vem com Python, portanto, é provável que você vá precisar fazer download e instalá-lo e, então, fazer download e instalar um editor de texto.

Instalando Python

Em primeiro lugar, verifique se Python está instalado em seu sistema. Abra uma janela de comando digitando **`command`** no menu Start (Iniciar) ou segurando a tecla SHIFT, ao mesmo tempo em que dá um clique com o botão direito do mouse em seu desktop, e selecione **Open command window here** (Abrir janela de comando aqui). Na janela do terminal, digite **`python`** com letras minúsculas. Se você obtiver um prompt Python (`>>>`), é sinal de que Python está instalado em seu sistema. No entanto, é provável que você vá ver uma mensagem de erro, informando que `python` não é um comando reconhecido.

Nesse caso, faça download do instalador de Python para Windows acessando *http://python.org/downloads/*. Você verá dois botões, um para fazer download de Python 3 e outro para download de Python 2. Clique no botão para Python 3, o que deverá iniciar automaticamente o download do instalador correto para o seu sistema. Depois de baixar o arquivo, execute o instalador. Certifique-se de marcar a opção Add Python to PATH (Adicionar Python ao PATH), que facilitará configurar o seu sistema corretamente. A Figura 1.2 mostra essa opção marcada.

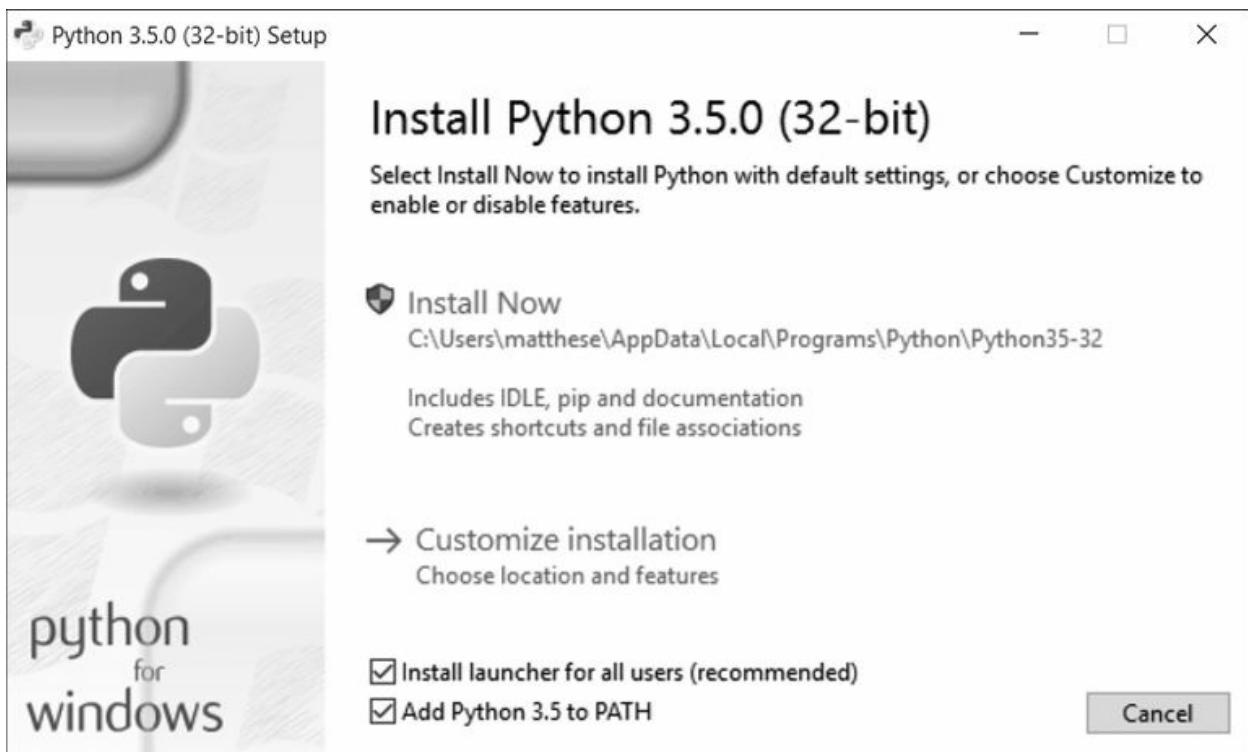



Figura 1.2 – Lembre-se de marcar a caixa identificada com Add Python to PATH (Acrescentar Python ao PATH).

Iniciando uma sessão de terminal Python

Configurar seu editor de texto será simples se você configurar antes o seu sistema para executar Python em uma sessão de terminal. Abra uma janela de comandos e digite **python** com letras minúsculas. Se você obtiver o prompt de Python (>>>), é sinal de que o Windows encontrou a versão de Python que você acabou de instalar:

```
C:\> python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Se esse comando funcionou, você poderá passar para a próxima seção, “Executando Python em uma sessão de terminal”.

No entanto, talvez você veja uma saída mais parecida com esta:

```
C:\> python
```

```
'python' is not recognized as an internal or external command, operable
program or batch file.
```

Nesse caso, precisamos informar como o Windows poderá encontrar a versão de Python que acabamos de instalar. O comando `python` de seu sistema normalmente é salvo em seu drive *C*, portanto, abra o Windows Explorer e acesse esse drive. Procure uma pasta que começa com o nome *Python*, abra-a e localize o arquivo *python* (com letras minúsculas). Por exemplo, eu tenho uma pasta *Python35* com um arquivo chamado *python*, portanto, o path para o comando `python` em meu sistema é *C:\Python35\python*. Se não encontrar esse arquivo, digite **`python`** na caixa de pesquisa do Windows Explorer para mostrar exatamente em que lugar o comando `python` está armazenado em seu sistema.

Quando achar que sabe qual é o path, teste-o fornecendo esse path em uma janela do terminal. Abra uma janela de comandos e digite o path completo que você acabou de encontrar:

```
C:\> C:\Python35\python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Se isso funcionar, você saberá como acessar Python em seu sistema.

Executando Python em uma sessão de terminal

Forneça a linha a seguir em sua sessão de Python e certifique-se de que verá a saída *Hello Python world!*

```
>>> print("Hello Python world!")
Hello Python world!
>>>
```

Sempre que quiser executar um trecho de código Python, abra uma janela de comandos e inicie uma sessão de terminal de Python. Para fechar a sessão de terminal, pressione CTRL-Z e então ENTER, ou digite o comando `exit()`.

Instalando um editor de texto

O Geany é um editor de texto simples: é fácil de instalar, permitirá executar quase todos os seus programas diretamente do editor em vez do terminal, dá destaque à sintaxe para colorir seu código e executa-o em uma janela do terminal; desse modo, você se acostumará a usar terminais. O Apêndice B tem informações sobre outros editores de texto, porém, recomendo usar o Geany, a menos que você tenha um bom motivo para utilizar um editor diferente.

Você pode fazer download de um instalador do Geany para Windows a partir de *http://geany.org/*. Clique em **Releases** (Versões) no menu Download e procure o instalador *geany-1.25_setup.exe* ou algo semelhante. Execute o instalador e aceite todos os defaults.

Para iniciar seu primeiro programa, abra o Geany: pressione a tecla Windows e procure o Geany em seu sistema. Crie um atalho arrastando o ícone para sua barra de tarefas ou o desktop. Crie uma pasta chamada *python_work* para seus projetos em algum lugar de seu sistema. (É melhor usar letras minúsculas e underscores para espaços em nomes de arquivos e de pastas, pois fazem parte das convenções de nomenclatura de Python.) Volte para o Geany e salve um arquivo Python vazio (**File ▸ Save As**, ou Arquivo ▸ Salvar como) chamado *hello_world.py* em sua pasta *python_work*. A extensão *.py* informa ao Geany que seu arquivo conterá um programa Python. Também diz ao Geany como executar seu programa e dar destaque ao texto de modo conveniente.

Após ter salvado seu arquivo, insira a linha a seguir:

```
print("Hello Python world!")
```

Se o comando `python` funcionou em seu sistema, não será necessário configurar o Geany; vá direto para a próxima seção e prossiga para “Executando o programa Hello Word”. Se você precisou fornecer um path como *C:\Python35\python* para iniciar um interpretador Python, siga as instruções da próxima seção para configurar o Geany em seu sistema.

Configurando o Geany

Para configurar o Geany, acesse **Build ▸ Set Build Commands** (Construir ▸ Definir Comandos de Construção). Você deverá ver as

palavras *Compile* e *Execute* com um comando ao lado de cada um. Os comandos Compile e Execute começam com `python` em letras minúsculas, mas o Geany não sabe em que lugar seu sistema armazenou o comando `python`. É preciso acrescentar o path usado na sessão de terminal.

Nos comandos Compile e Execute, acrescente o drive em que está o seu comando `python` e a pasta em que esse comando está armazenado. Seu comando Compile deve ter o seguinte aspecto:

```
C:\Python35\python -m py_compile "%f"
```

Seu path pode ser um pouco diferente, mas certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas estejam iguais ao que foi mostrado aqui.

Seu comando Execute deve ter o seguinte aspecto:

```
C:\Python35\python "%f"
```

Novamente, certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas em seu comando Execute estejam iguais ao que foi mostrado aqui. A Figura 1.3 mostra como devem ser esses comandos no menu de configuração do Geany.

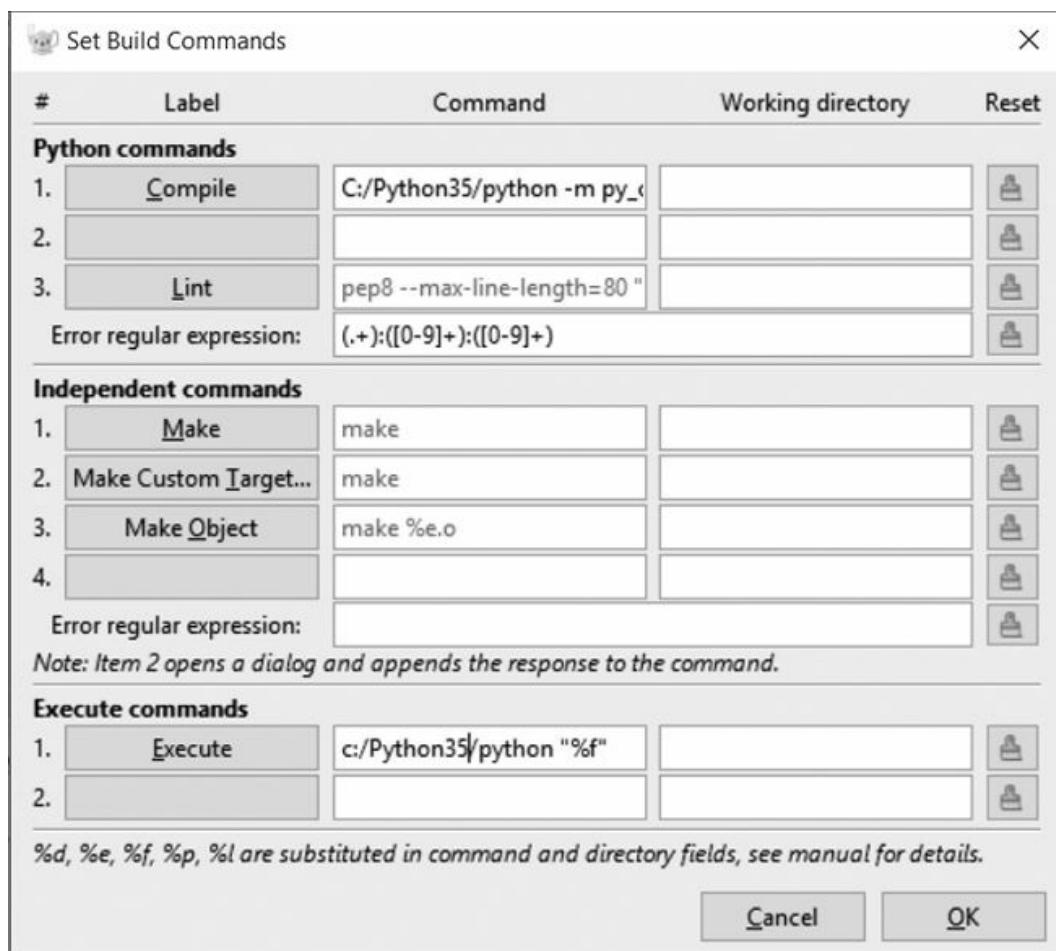



Figura 1.3 – Nesse caso, o Geany está configurado para usar Python 3 no Windows.

Após ter configurado esses comandos corretamente, clique em **OK**.

Executando o programa Hello World

Agora você deverá ser capaz de executar seu programa com sucesso. Execute *hello_world.py* selecionando **Build ▸ Execute** (Construir ▸ Executar) no menu, clicando no ícone Execute (Executar, que mostra um conjunto de engrenagens) ou pressionando F5. Uma janela de terminal deve aparecer com a saída a seguir:

```
Hello Python world!

------------------
(program exited with code: 0)
Press return to continue
```

Se não vir isso, confira todos os caracteres da linha que você inseriu. Você

não usou uma letra maiúscula acidentalmente em `print`? Você não esqueceu uma das aspas, ou as duas, ou os parênteses? As linguagens de programação esperam uma sintaxe bem específica e, se você não a fornecer, haverá erros. Se não conseguir fazer o programa executar, consulte a próxima seção para obter ajuda.

Resolvendo problemas de instalação

Esperamos que a configuração de seu ambiente de programação tenha sido um sucesso, mas se você não conseguiu executar *hello_world.py*, apresentamos a seguir algumas alternativas que você poderá experimentar:

- Quando um programa contiver um erro significativo, Python exibirá um *traceback*. O interpretador olha o arquivo e tenta informar o problema. O traceback pode oferecer uma pista sobre o problema que está impedindo seu programa de executar.
- Afaste-se de seu computador, faça um pequeno intervalo e, então, tente novamente. Lembre-se de que a sintaxe é muito importante em programação, de modo que, até mesmo a falta de um caractere de dois-pontos, uma aspa ausente ou um parêntese sem par pode impedir que um programa execute de forma apropriada. Releia as partes relevantes deste capítulo, verifique o que você fez e veja se consegue encontrar o erro.
- Comece de novo. Provavelmente você não precisará desinstalar nada, mas pode ser que faça sentido apagar seu arquivo *hello_world.py* e criá-lo novamente do zero.
- Peça a outra pessoa para seguir os passos deste capítulo em seu computador, ou em um computador diferente, e observe essa pessoa atentamente. Talvez você tenha deixado passar um pequeno passo que outra pessoa, por acaso, percebeu.
- Encontre alguém que conheça Python e peça-lhe ajuda. Se perguntar por aí, talvez você descubra que conhece alguém que use Python.
- As instruções de configuração deste capítulo também estão disponíveis online em *https://www.nostarch.com/pythoncrashcourse/*. A versão online dessas instruções talvez seja mais adequada a você.

- Peça ajuda online. O Apêndice C apresenta vários recursos e áreas online, como fóruns e sites com bate-papos ao vivo, em que é possível pedir soluções para pessoas que já passaram pelo problema que você está enfrentando no momento.

Não se preocupe em incomodar programadores experientes. Todo programador já ficou sem saber o que fazer em algum momento, e a maioria deles ficará feliz em ajudar você a configurar seu sistema corretamente. Desde que você consiga explicar claramente o que está tentando fazer, o que já tentou e os resultados que está obtendo, há uma boa chance de que alguém possa ajudá-lo. Como mencionamos na Introdução, a comunidade Python é bem receptiva aos iniciantes.

Python deve executar bem em qualquer computador moderno, portanto, encontre uma maneira de pedir ajuda se estiver com problemas até o momento. Problemas iniciais podem ser frustrantes, mas vale a pena resolvê-los. Depois que conseguir executar o programa *hello_world.py*, você poderá começar a aprender Python e seu trabalho de programação se tornará mais interessante e satisfatório.

Executando programas Python a partir de um terminal

A maior parte dos programas que você escrever em seu editor de texto será executada diretamente do editor, mas, às vezes, será conveniente executar programas a partir de um terminal. Por exemplo, talvez você queira executar um programa existente sem abri-lo para edição.

Você pode fazer isso em qualquer sistema com Python instalado se souber como acessar o diretório em que seu arquivo de programa está armazenado. Para testar isso, certifique-se de ter salvo o arquivo *hello_world.py* na pasta *python_work* em seu desktop.

No Linux e no OS X

Executar um programa Python a partir de uma sessão de terminal é igual no Linux e no OS X. O comando de terminal `cd`, de *change directory* (mudar diretório), é usado para navegar pelo seu sistema de arquivos em uma sessão de terminal. O comando `ls`, de *list* (listar), mostra todos os

arquivos não ocultos do diretório atual.

Abra uma nova janela de terminal e dê os comandos a seguir para executar *hello_world.py*:

```
u ~$ cd Desktop/python_work/
v ~/Desktop/python_work$ ls
  hello_world.py
w ~/Desktop/python_work$ python hello_world.py
  Hello Python world!
```

Em u usamos `cd` para navegar até a pasta *python_work*, que está na pasta *Desktop*. Em seguida, utilizamos `ls` para garantir que *hello_world.py* está nessa pasta v. Então executamos o arquivo usando o comando `python hello_world.py` w.

É simples assim. Basta usar o comando `python` (ou `python3`) para executar programas Python.

No Windows

O comando de terminal `cd`, de *change directory* (mudar diretório), é usado para navegar pelo seu sistema de arquivos em uma sessão de terminal. O comando `dir`, de *directory* (diretório), mostra todos os arquivos do diretório atual.

Abra uma nova janela e dê os comandos a seguir para executar *hello_world.py*:

```
u C:\> cd Desktop\python_work
v C:\Desktop\python_work> dir
  hello_world.py
w C:\Desktop\python_work> python hello_world.py
  Hello Python world!
```

Em u usamos o comando `cd` para navegar até a pasta *python_work*, que está na pasta Desktop. Em seguida, utilizamos o comando `dir` para garantir que *hello_world.py* está nessa pasta v. Então executamos o arquivo usando o comando `python hello_world.py` w.

Se você não havia configurado seu sistema para usar o comando simples `python`, talvez seja necessário usar a versão mais longa desse comando:

```
C:\$ cd Desktop\python_work
C:\Desktop\python_work$ dir
hello_world.py
C:\Desktop\python_work$ C:\Python35\python hello_world.py
Hello Python world!
```

A maior parte de seus programas executará de forma apropriada, diretamente de seu editor, mas, à medida que seu trabalho se tornar mais complexo, você poderá escrever programas que precisarão ser executados a partir de um terminal.

FAÇA VOCÊ MESMO

Os exercícios deste capítulo são exploratórios por natureza. A partir do Capítulo 2, os desafios que você resolverá serão baseados no que você aprendeu.

1.1 – python.org: Explore a página inicial de Python (*http://python.org/*) para encontrar tópicos pelos quais você se interesse. À medida que tiver mais familiaridade com Python, diferentes partes do site serão mais úteis a você.

1.2 – Erros de digitação em Hello World: Abra o arquivo *hello_world.py* que você acabou de criar. Introduza um erro de digitação em algum ponto da linha e execute o programa novamente. Você é capaz de inserir um erro de digitação que gere um erro? Consegue entender a mensagem de erro? É capaz de inserir um erro de digitação que não produza um erro? Por que você acha que isso não gerou um erro?

1.3 – Habilidades infinitas: Se tivesse infinitas habilidades de programação, o que você desenvolveria? Você está prestes a aprender a programar. Se tiver um objetivo final em mente, você terá um uso imediato para suas novas habilidades; agora é um ótimo momento para esboçar descrições do que você gostaria de criar. É um bom hábito manter um caderno com "ideias" que você possa consultar sempre que quiser iniciar um novo projeto. Reserve alguns minutos para descrever três programas que você gostaria de criar.

Resumo

Neste capítulo aprendemos um pouco sobre Python em geral, e instalamos Python em seu sistema caso ainda não estivesse presente. Também instalamos um editor de texto para facilitar escrever código Python. Aprendemos a executar trechos de código Python em uma sessão de terminal e executamos nosso primeiro programa, *hello_world.py*. É

provável que você também tenha aprendido um pouco sobre resolução de problemas.

No próximo capítulo conheceremos diferentes tipos de dados com os quais podemos trabalhar em nossos programas Python e aprenderemos a usar variáveis também.

2
VARIÁVEIS E TIPOS DE DADOS SIMPLES

Neste capítulo conheceremos os diferentes tipos de dados com os quais podemos trabalhar em nossos programas Python. Aprenderemos também a armazenar dados em variáveis e usar essas variáveis em nossos programas.

O que realmente acontece quando executamos hello_world.py

Vamos observar com mais detalhes o que Python faz quando executamos *hello_world.py*. O fato é que Python realiza muitas tarefas, mesmo quando executa um programa simples:

hello_world.py

```
print("Hello Python world!")
```

Quando executar esse código, você deverá ver a saída a seguir:

```
Hello Python world!
```

Ao executar *hello_world.py*, o arquivo terminado em *.py* indica que é um programa Python. Seu editor então passa o programa pelo *interpretador Python*, que lê o programa todo e determina o que cada palavra significa. Por exemplo, quando vê a palavra `print`, o interpretador exibe na tela o que quer que esteja dentro dos parênteses.

À medida que você escrever seus programas, o editor dará destaque a diferentes partes do código de modo distinto. Por exemplo, ele reconhece que `print` é o nome de uma função e exibe essa palavra em azul. Reconhece que "Hello Python world!" não é um código Python e exibe essa frase em laranja. Esse recurso se chama *destaque de sintaxe* e é muito útil quando você começar a escrever seus próprios programas.

Variáveis

Vamos experimentar usar uma variável em *hello_world.py*. Acrescente uma nova linha no início do arquivo e modifique a segunda linha:

```
message = "Hello Python world!"
print(message)
```

Execute esse programa para ver o que acontece. Você deverá ver a mesma saída vista anteriormente:

```
Hello Python world!
```

Acrescentamos uma *variável* chamada `message`. Toda variável armazena um *valor*, que é a informação associada a essa variável. Nesse caso, o valor é o texto “Hello Python world!”.

Acrescentar uma variável implica um pouco mais de trabalho para o interpretador Python. Quando processa a primeira linha, ele associa o texto “Hello Python world!” à variável `message`. Quando chega à segunda linha, o interpretador exibe o valor associado à `message` na tela.

Vamos expandir esse programa modificando *hello_world.py* para que exiba uma segunda mensagem. Acrescente uma linha em branco em *hello_world.py* e, em seguida, adicione duas novas linhas de código:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Ao executar *hello_world.py* agora você deverá ver duas linhas na saída:

```
Hello Python world!
Hello Python Crash Course world!
```

Podemos mudar o valor de uma variável em nosso programa a qualquer momento, e Python sempre manterá o controle do valor atual.

Nomeando e usando variáveis

Ao usar variáveis em Python, é preciso seguir algumas regras e diretrizes. Quebrar algumas dessas regras provocará erros; outras diretrizes simplesmente ajudam a escrever um código mais fácil de ler e de entender. Lembre-se de ter as seguintes regras em mente:

- Nomes de variáveis podem conter apenas letras, números e

underscores. Podem começar com uma letra ou um underscore, mas não com um número. Por exemplo, podemos chamar uma variável de *message_1*, mas não de *1_message*.

- Espaços não são permitidos em nomes de variáveis, mas underscores podem ser usados para separar palavras em nomes de variáveis. Por exemplo, *greeting_message* funciona, mas *greeting message* causará erros.
- Evite usar palavras reservadas e nomes de funções em Python como nomes de variáveis, ou seja, não use palavras que Python reservou para um propósito particular de programação, por exemplo, a palavra `print`. (Veja a seção "Palavras reservadas e funções embutidas de Python".)
- Nomes de variáveis devem ser concisos, porém descritivos. Por exemplo, *name* é melhor que *n*, *student_name* é melhor que *s_n* e *name_length* é melhor que *length_of_persons_name*.
- Tome cuidado ao usar a letra minúscula *l* e a letra maiúscula *O*, pois elas podem ser confundidas com os números *1* e *0*.

Aprender a criar bons nomes de variáveis, em especial quando seus programas se tornarem mais interessantes e complicados, pode exigir um pouco de prática. À medida que escrever mais programas e começar a ler códigos de outras pessoas, você se tornará mais habilidoso na criação de nomes significativos.

NOTA As variáveis Python que você está usando no momento devem utilizar letras minúsculas. Você não terá erros se usar letras maiúsculas, mas é uma boa ideia evitá-las por enquanto.

Evitando erros em nomes ao usar variáveis

Todo programador comete erros, e a maioria comete erros todos os dias. Embora bons programadores possam criar erros, eles também sabem responder a esses erros de modo eficiente. Vamos observar um erro que você provavelmente cometerá no início e vamos aprender a corrigi-lo.

Escreveremos um código que gera um erro propositadamente. Digite o código a seguir, incluindo a palavra *mesage* com um erro de ortografia, conforme mostrada em negrito:

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

Quando houver um erro em seu programa, o interpretador Python faz o melhor que puder para ajudar você a descobrir onde está o problema. O interpretador oferece um traceback quando um programa não é capaz de executar com sucesso. Um *traceback* é um registro do ponto em que o interpretador se deparou com problemas quando tentou executar seu código. Eis um exemplo do traceback fornecido por Python após o nome da variável ter sido digitado incorretamente por engano:

```
  Traceback (most recent call last):
u   File "hello_world.py", line 2, in <module>
v     print(mesage)
w NameError: name 'mesage' is not defined
```

A saída em u informa que um erro ocorreu na linha 2 do arquivo *hello_world.py*. O interpretador mostra essa linha para nos ajudar a identificar o erro rapidamente v e informa o tipo do erro encontrado w. Nesse caso, ele encontrou um *erro de nome* e informa que a variável exibida, `mesage`, não foi definida. Python não é capaz de identificar o nome da variável especificada. Um erro de nome geralmente quer dizer que esquecemos de definir o valor de uma variável antes de usá-la ou que cometemos um erro de ortografia quando fornecemos o nome da variável.

É claro que, nesse exemplo, omitimos a letra *s* do nome da variável `message` na segunda linha. O interpretador Python não faz uma verificação de ortografia em seu código, mas garante que os nomes das variáveis estejam escritos de forma consistente. Por exemplo, observe o que acontece quando escrevemos *message* incorretamente em outro ponto do código também:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

Nesse caso, o programa executa com sucesso!

```
Hello Python Crash Course reader!
```

Os computadores são exigentes, mas não se preocupam com uma

ortografia correta ou incorreta. Como resultado, você não precisa levar em consideração a ortografia ou as regras gramaticais do inglês (ou de português) quando tentar criar nomes de variáveis e escrever código.

Muitos erros de programação são apenas erros de digitação de um caractere em uma linha de um programa. Se você gasta muito tempo procurando um desses erros, saiba que está em boa companhia. Muitos programadores experientes e talentosos passam horam caçando esses tipos de pequenos erros. Tente rir da situação e siga em frente, sabendo que isso acontecerá com frequência durante sua vida de programador.

NOTA A melhor maneira de entender novos conceitos de programação é tentar usá-los em seus programas. Se não souber o que fazer quando estiver trabalhando em um exercício deste livro, procure dedicar-se a outra tarefa por um tempo. Se continuar sem conseguir avançar, analise a parte relevante do capítulo. Se ainda precisar de ajuda, veja as sugestões no Apêndice C.

FAÇA VOCÊ MESMO

Escreva um programa separado para resolver cada um destes exercícios. Salve cada programa com um nome de arquivo que siga as convenções-padrão de Python, com letras minúsculas e underscores, por exemplo, *simple_message.py* e *simple_messages.py*.

2.1 – Mensagem simples: Armazene uma mensagem em uma variável e, em seguida, exiba essa mensagem.

2.2 – Mensagens simples: Armazene uma mensagem em uma variável e, em seguida, exiba essa mensagem. Então altere o valor de sua variável para uma nova mensagem e mostre essa nova mensagem.

Strings

Como a maior parte dos programas define e reúne algum tipo de dado e então faz algo útil com eles, classificar diferentes tipos de dados é conveniente. O primeiro tipo de dado que veremos é a string. As strings, à primeira vista, são bem simples, mas você pode usá-las de vários modos diferentes.

Uma *string* é simplesmente uma série de caracteres. Tudo que estiver entre aspas é considerada uma string em Python, e você pode usar aspas

simples ou duplas em torno de suas strings, assim:

```
"This is a string."
'This is also a string.'
```

Essa flexibilidade permite usar aspas e apóstrofos em suas strings:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Vamos explorar algumas das maneiras de usar strings.

Mudando para letras maiúsculas e minúsculas em uma string usando métodos

Uma das tarefas mais simples que podemos fazer com strings é mudar o tipo de letra, isto é, minúscula ou maiúscula, das palavras de uma string. Observe o código a seguir e tente determinar o que está acontecendo:

name.py

```
name = "ada lovelace"
print(name.title())
```

Salve esse arquivo como *name.py* e então execute-o. Você deverá ver esta saída:

```
Ada Lovelace
```

Nesse exemplo, a string com letras minúsculas `"ada lovelace"` é armazenada na variável `name`. O método `title()` aparece depois da variável na instrução `print()`. Um *método* é uma ação que Python pode executar em um dado. O ponto (.) após `name` em `name.title()` informa a Python que o método `title()` deve atuar na variável `name`. Todo método é seguido de um conjunto de parênteses, pois os métodos, com frequência, precisam de informações adicionais para realizar sua tarefa. Essas informações são fornecidas entre os parênteses. A função `title()` não precisa de nenhuma informação adicional, portanto seus parênteses estão vazios.

`title()` exibe cada palavra com uma letra maiúscula no início. Isso é útil,

pois, muitas vezes, você vai querer pensar em um nome como uma informação. Por exemplo, você pode querer que seu programa reconheça os valores de entrada `Ada`, `ADA` e `ada` como o mesmo nome e exiba todos eles como `Ada`.

Vários outros métodos úteis estão disponíveis para tratar letras maiúsculas e minúsculas também. Por exemplo, você pode mudar uma string para que tenha somente letras maiúsculas ou somente letras minúsculas, assim:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

Essas instruções exibirão o seguinte:

```
ADA LOVELACE
ada lovelace
```

O método `lower()` é particularmente útil para armazenar dados. Muitas vezes, você não vai querer confiar no fato de seus usuários fornecerem letras maiúsculas ou minúsculas, portanto fará a conversão das strings para letras minúsculas antes de armazená-las. Então, quando quiser exibir a informação, usará o tipo de letra que fizer mais sentido para cada string.

Combinando ou concatenando strings

Muitas vezes, será conveniente combinar strings. Por exemplo, você pode querer armazenar um primeiro nome e um sobrenome em variáveis separadas e, então, combiná-las quando quiser exibir o nome completo de alguém:

```
  first_name = "ada"
  last_name = "lovelace"
u full_name = first_name + " " + last_name

  print(full_name)
```

Python usa o símbolo de adição (`+`) para combinar strings. Nesse exemplo, usamos `+` para criar um nome completo combinando `first_name`,

um espaço e `last_name` u, o que resultou em:

```
ada lovelace
```

Esse método de combinar strings se chama *concatenação*. Podemos usar concatenação para compor mensagens completas usando informações armazenadas em uma variável. Vamos observar um exemplo:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

u print("Hello, " + full_name.title() + "!")
```

Nesse caso, o nome completo é usado em u, em uma sentença que saúda o usuário, e o método `title()` é utilizado para formatar o nome de forma apropriada. Esse código devolve uma saudação simples, porém formatada de modo elegante:

```
Hello, Ada Lovelace!
```

Podemos usar concatenação para compor uma mensagem e então armazenar a mensagem completa em uma variável:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

u message = "Hello, " + full_name.title() + "!"
v print(message)
```

Esse código também exibe a mensagem "Hello, Ada Lovelace!", mas armazenar a mensagem em uma variável em u deixa a instrução `print` final em v muito mais simples.

Acrescentando espaços em branco em strings com tabulações ou quebras de linha

Em programação, *espaços em branco* se referem a qualquer caractere que não é mostrado, como espaços, tabulações e símbolos de fim de linha. Podemos usar espaços em branco para organizar a saída de modo que ela

seja mais legível aos usuários.

Para acrescentar uma tabulação em seu texto, utilize a combinação de caracteres `\t` como mostrado em u:

```
  >>> print("Python")
  Python
u >>> print("\tPython")
      Python
```

Para acrescentar uma quebra de linha em uma string, utilize a combinação de caracteres `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

Também podemos combinar tabulações e quebras de linha em uma única string. A string `"\n\t"` diz a Python para passar para uma nova linha e iniciar a próxima com uma tabulação. O exemplo a seguir mostra como usar uma string de uma só linha para gerar quatro linhas na saída:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Quebras de linha e tabulações serão muito úteis nos próximos dois capítulos, quando começaremos a gerar muitas linhas de saída usando apenas algumas linhas de código.

Removendo espaços em branco

Espaços em branco extras podem ser confusos em seus programas. Para os programadores, `'python'` e `'python '` parecem praticamente iguais. Contudo, para um programa, são duas strings diferentes. Python identifica o espaço extra em `'python '` e o considera significativo, a menos que você informe o contrário.

É importante pensar em espaços em branco, pois, com frequência, você vai querer comparar duas strings para determinar se são iguais. Por exemplo, uma situação importante pode envolver a verificação dos nomes de usuário das pessoas quando elas fizerem login em um site. Espaços em branco extras podem ser confusos em situações muito mais simples também. Felizmente, Python facilita eliminar espaços em branco extras dos dados fornecidos pelas pessoas.

Python é capaz de encontrar espaços em branco dos lados direito e esquerdo de uma string. Para garantir que não haja espaços em branco do lado direito de uma string, utilize o método `rstrip()`.

```
u >>> favorite_language = 'python '
v >>> favorite_language
  'python '
w >>> favorite_language.rstrip()
  'python'
x >>> favorite_language
  'python '
```

O valor armazenado em `favorite_language` em u contém um espaço em branco extra no final da string. Quando solicitamos esse valor a Python em uma sessão de terminal, podemos ver o espaço no final do valor v. Quando o método `rstrip()` atua na variável `favorite_language` em w, esse espaço extra é removido. Entretanto, a remoção é temporária. Se solicitar o valor de `favorite_language` novamente, você poderá ver que a string é a mesma que foi fornecida, incluindo o espaço em branco extra x.

Para remover o espaço em branco da string de modo permanente, você deve armazenar o valor com o caractere removido de volta na variável:

```
  >>> favorite_language = 'python '
u >>> favorite_language = favorite_language.rstrip()
  >>> favorite_language
  'python'
```

Para remover o espaço em branco da string, você deve remover o espaço em branco do lado direito da string e então armazenar esse valor de volta na variável original, como mostrado em u. Alterar o valor de uma variável e então armazenar o novo valor de volta na variável original é uma operação

frequente em programação. É assim que o valor de uma variável pode mudar à medida que um programa é executado ou em resposta à entrada de usuário.

Também podemos remover espaços em branco do lado esquerdo de uma string usando o método `lstrip()`, ou remover espaços em branco dos dois lados ao mesmo tempo com `strip()`:

```
u >>> favorite_language = ' python '
v >>> favorite_language.rstrip()
  ' python'
w >>> favorite_language.lstrip()
  'python '
x >>> favorite_language.strip()
  'python'
```

Nesse exemplo, começamos com um valor que tem espaços em branco no início e no fim u. Então removemos os espaços extras do lado direito em v, do lado esquerdo em w e de ambos os lados em x. Fazer experimentos com essas funções de remoção pode ajudar você a ter familiaridade com a manipulação de strings. No mundo real, essas funções de remoção são usadas com mais frequência para limpar entradas de usuário antes de armazená-las em um programa.

Evitando erros de sintaxe com strings

Um tipo de erro que você poderá ver com certa regularidade é um erro de sintaxe. Um *erro de sintaxe* ocorre quando Python não reconhece uma seção de seu programa como um código Python válido. Por exemplo, se usar um apóstrofo entre aspas simples, você produzirá um erro. Isso acontece porque Python interpreta tudo que estiver entre a primeira aspa simples e o apóstrofo como uma string. Ele então tenta interpretar o restante do texto como código Python, o que causa erros.

Eis o modo de usar aspas simples e duplas corretamente. Salve este programa como *apostrophe.py* e então execute-o:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
```

```
print(message)
```

O apóstrofo aparece entre um conjunto de aspas duplas, portanto o interpretador Python não terá nenhum problema para ler a string corretamente:

```
One of Python's strengths is its diverse community.
```

No entanto, se você usar aspas simples, o interpretador Python não será capaz de identificar em que ponto a string deve terminar:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

Você verá a saída a seguir:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
                            ^u
SyntaxError: invalid syntax
```

Na saída, podemos ver que o erro ocorre em `u`, logo depois da segunda aspa simples. Esse *erro de sintaxe* informa que o interpretador não reconheceu algo como código Python válido. Os erros podem ter origens variadas, e destacarei alguns erros comuns à medida que surgirem. Você poderá ver erros de sintaxe com frequência enquanto aprende a escrever código Python apropriado. Erros de sintaxe também são o tipo menos específico de erro, portanto podem ser difíceis e frustrantes para identificar e corrigir. Se você não souber o que fazer quanto a um erro particularmente persistente, consulte as sugestões no Apêndice C.

NOTA O recurso de destaque de sintaxe de seu editor deve ajudar você a identificar alguns erros de sintaxe rapidamente quando escrever seus programas. Se você vir código Python em destaque como se fosse inglês, ou inglês destacado como se fosse código Python, é provável que haja uma aspa sem correspondente em algum ponto de seu arquivo.

Exibindo informações em Python 2

A instrução `print` tem uma sintaxe levemente diferente em Python 2:

```
>>> python2.7
```

```
>>> print "Hello Python 2.7 world!"
Hello Python 2.7 world!
```

Os parênteses não são necessários em torno de frases que você quer exibir em Python 2. Tecnicamente, `print` é uma função em Python 3, motivo pelo qual os parênteses são necessários. Algumas instruções `print` em Python 2 incluem parênteses, mas o comportamento pode ser um pouco diferente do que você verá em Python 3. Basicamente, quando estiver vendo código escrito em Python 2, espere ver algumas instruções `print` com parênteses e outras sem.

FAÇA VOCÊ MESMO

Salve cada um dos exercícios a seguir em um arquivo separado com um nome como *name_cases.py*. Se não souber o que fazer, descase um pouco ou consulte as sugestões que estão no Apêndice C.

2.3 – Mensagem pessoal: Armazene o nome de uma pessoa em uma variável e apresente uma mensagem a essa pessoa. Sua mensagem deve ser simples, como "Alô Eric, você gostaria de aprender um pouco de Python hoje?".

2.4 – Letras maiúsculas e minúsculas em nomes: Armazene o nome de uma pessoa em uma variável e então apresente o nome dessa pessoa em letras minúsculas, em letras maiúsculas e somente com a primeira letra maiúscula.

2.5 – Citação famosa: Encontre uma citação de uma pessoa famosa que você admire. Exiba a citação e o nome do autor. Sua saída deverá ter a aparência a seguir, incluindo as aspas:

Albert Einstein certa vez disse: "Uma pessoa que nunca cometeu um erro jamais tentou nada novo."

2.6 – Citação famosa 2: Repita o Exercício 2.5, porém, desta vez, armazene o nome da pessoa famosa em uma variável chamada `famous_person`. Em seguida, componha sua mensagem e armazene-a em uma nova variável chamada `message`. Exiba sua mensagem.

2.7 – Removendo caracteres em branco de nomes: Armazene o nome de uma pessoa e inclua alguns caracteres em branco no início e no final do nome. Lembre-se de usar cada combinação de caracteres, `"\t"` e `"\n"`, pelo menos uma vez.

Exiba o nome uma vez, de modo que os espaços em branco em torno do nome sejam mostrados. Em seguida, exiba o nome usando cada uma das três funções de remoção de espaços: `lstrip()`, `rstrip()` e `strip()`.

Números

Os números são usados com muita frequência em programação para armazenar pontuações em jogos, representar dados em visualizações, guardar informações em aplicações web e assim por diante. Python trata números de várias maneiras diferentes, de acordo com o modo como são usados. Vamos analisar inicialmente como Python trata inteiros, pois eles são os dados mais simples para trabalhar.

Inteiros

Você pode somar (+), subtrair (-), multiplicar (*) e dividir (/) inteiros em Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Em uma sessão de terminal, Python simplesmente devolve o resultado da operação. Dois símbolos de multiplicação são usados em Python para representar exponenciais:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

A linguagem Python também aceita a ordem das operações, portanto você pode fazer várias operações em uma expressão. Também podemos usar parênteses para modificar a ordem das operações para que Python possa avaliar sua expressão na ordem que você especificar. Por exemplo:

```
>>> 2 + 3*4
```

```
14
>>> (2 + 3) * 4
20
```

Os espaços nesses exemplos não têm nenhum efeito no modo como Python avalia as expressões: eles simplesmente ajudam a identificar mais rapidamente as operações que têm prioridade quando lemos o código.

Números de ponto flutuante

Python chama qualquer número com um ponto decimal de *número de ponto flutuante* (float). Esse termo é usado na maioria das linguagens de programação e refere-se ao fato de um ponto decimal poder aparecer em qualquer posição em um número. Toda linguagem de programação deve ser cuidadosamente projetada para lidar de modo adequado com números decimais de modo que eles se comportem de forma apropriada, não importa em que lugar o ponto decimal apareça.

Na maioria das ocasiões, podemos usar decimais sem nos preocupar com o modo como eles se comportam. Basta fornecer os números que você quer usar, e Python provavelmente fará o que você espera:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

No entanto, tome cuidado, pois, às vezes, você poderá obter um número arbitrário de casas decimais em sua resposta:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Isso acontece em todas as linguagens e não é motivo para muita preocupação. Python tenta encontrar uma forma de representar o resultado

do modo mais exato possível, o que, às vezes, é difícil, considerando a maneira como os computadores devem representar os números internamente. Basta ignorar as casas decimais extras por enquanto; você aprenderá a lidar com casas extras quando for necessário nos projetos da Parte II.

Evitando erros de tipo com a função str()

Com frequência, você vai querer usar o valor de uma variável em uma mensagem. Por exemplo, suponha que você queira desejar feliz aniversário a alguém. Você poderia escrever um código como este:

birthday.py

```
age = 23
message = "Happy " + age + "rd Birthday!"

print(message)
```

Você esperaria que esse código exibisse a seguinte saudação simples de feliz aniversário: `Happy 23rd birthday!`. Contudo, se executar esse código, verá que ele gera um erro:

```
  Traceback (most recent call last):
    File "birthday.py", line 2, in <module>
      message = "Happy " + age + "rd Birthday!"
u TypeError: Can't convert 'int' object to str implicitly
```

É um *erro de tipo*. Significa que Python não é capaz de reconhecer o tipo de informação que você está usando. Nesse exemplo, Python vê que você está usando uma variável em **u** cujo valor é um inteiro (`int`), mas não tem certeza de como interpretar esse valor. O interpretador sabe que a variável poderia representar um valor numérico 23 ou os caracteres *2* e *3*. Quando usar inteiros em strings desse modo, você precisará especificar explicitamente que quer que Python utilize o inteiro como uma string de caracteres. Podemos fazer isso envolvendo a variável com a função `str()`; essa função diz a Python para representar valores que não são strings como strings:

```
age = 23
message = "Happy " + str(age) + "rd Birthday!"

print(message)
```

Com isso, Python sabe que você quer converter o valor numérico 23 em uma string e exibir os caracteres `2` e `3` como parte da mensagem de feliz aniversário. Agora você obterá a mensagem esperada, sem erros:

```
Happy 23rd Birthday!
```

Trabalhar com números em Python é simples na maior parte do tempo. Se você estiver obtendo resultados inesperados, verifique se Python está interpretando seus números da forma desejada, seja como um valor numérico, seja como um valor de string.

Inteiros em Python 2

Python 2 devolve um resultado um pouco diferente quando dividimos dois inteiros:

```
>>> python2.7
>>> 3 / 2
1
```

Em vez de `1.5`, Python devolve `1`. A divisão de inteiros em Python 2 resulta em um inteiro com o resto truncado. Observe que o resultado não é um inteiro arredondado; o resto é simplesmente omitido.

Para evitar esse comportamento em Python 2, certifique-se de que pelo menos um dos números seja um número de ponto flutuante. Ao fazer isso, o resultado também será um número de ponto flutuante:

```
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> 3 / 2.0
1.5
>>> 3.0 / 2.0
1.5
```

Esse comportamento da divisão é uma fonte comum de confusão quando as pessoas que estão acostumadas a usar Python 3 começam a usar Python 2 ou vice-versa. Se você usa ou cria código que mistura inteiros e números de ponto flutuante, tome cuidado com comportamentos irregulares.

FAÇA VOCÊ MESMO

2.8 – Número oito: Escreva operações de adição, subtração, multiplicação e divisão que resultem no número 8. Lembre-se de colocar suas operações em instruções `print` para ver os resultados. Você deve criar quatro linhas como esta:

```
print(5 + 3)
```

Sua saída deve simplesmente ser composta de quatro linhas, com o número 8 em cada uma das linhas.

2.9 – Número favorito: Armazene seu número favorito em uma variável. Em seguida, usando essa variável, crie uma mensagem que revele o seu número favorito. Exiba essa mensagem.

Comentários

Comentários são um recurso extremamente útil na maioria das linguagens de programação. Tudo que você escreveu em seus programas até agora é código Python. À medida que seus programas se tornarem mais longos e complicados, você deve acrescentar notas que descrevam a abordagem geral adotada para o problema que você está resolvendo. Um comentário permite escrever notas em seus programas em linguagem natural.

Como escrever comentários?

Em Python, o caractere sustenido (#) indica um comentário. Tudo que vier depois de um caractere sustenido em seu código será ignorado pelo interpretador Python. Por exemplo:

comment.py

```
# Diga olá a todos
print("Hello Python people!")
```

Python ignora a primeira linha e executa a segunda.

```
Hello Python people!
```

Que tipos de comentário você deve escrever?

O principal motivo para escrever comentários é explicar o que seu código deve fazer e como você o faz funcionar. Quando estiver no meio do trabalho de um projeto, você entenderá como todas as partes se encaixam. Porém, quando retomar um projeto depois de passar um tempo afastado, é provável que você vá esquecer alguns detalhes. É sempre possível estudar seu código por um tempo e descobrir como os segmentos deveriam funcionar, mas escrever bons comentários pode fazer você economizar tempo ao sintetizar sua abordagem geral em linguagem natural clara.

Se quiser se tornar um programador profissional ou colaborar com outros programadores, escreva comentários significativos. Atualmente, a maior parte dos softwares é escrita de modo colaborativo, seja por um grupo de funcionários em uma empresa, seja por um grupo de pessoas que trabalham juntas em um projeto de código aberto. Programadores habilidosos esperam ver comentários no código, portanto é melhor começar a adicionar comentários descritivos em seus programas agora. Escrever comentários claros e concisos em seu código é um dos hábitos mais benéficos que você pode desenvolver como um novo programador.

Quando estiver decidindo se deve escrever um comentário, pergunte a si mesmo se você precisou considerar várias abordagens antes de definir uma maneira razoável de fazer algo funcionar; em caso afirmativo, escreva um comentário sobre sua solução. É muito mais fácil apagar comentários extras depois que retornar e escrever comentários em um programa pouco comentado. A partir de agora, usarei comentários em exemplos deste livro para ajudar a explicar algumas seções de código.

FAÇA VOCÊ MESMO

2.10 – Acrescentando comentários: Escolha dois dos programas que você escreveu e acrescente pelo menos um comentário em cada um. Se você não tiver nada específico para escrever porque o programa é muito simples no momento, basta adicionar seu nome e a data de hoje no início de cada arquivo de programa. Em seguida, escreva uma frase que descreva o que o programa faz.

Zen de Python

Durante muito tempo, a linguagem de programação Perl foi o principal sustentáculo da internet. A maioria dos primeiros sites interativos usava scripts Perl. O lema da comunidade Perl na época era: "Há mais de uma maneira de fazer algo". As pessoas gostaram dessa postura por um tempo porque a flexibilidade inscrita na linguagem possibilitava resolver a maior parte dos problemas de várias maneiras. Essa abordagem era aceitável enquanto trabalhávamos em nossos próprios projetos, mas, em algum momento, as pessoas perceberam que a ênfase na flexibilidade dificultava a manutenção de projetos grandes em longo prazo. Revisar código e tentar descobrir o que outra pessoa pensou quando resolvia um problema complexo era difícil, tedioso e consumia bastante tempo.

Programadores Python experientes incentivarão você a evitar a complexidade e buscar a simplicidade sempre que for possível. A filosofia da comunidade Python está contida no "Zen de Python" de Tim Peters. Você pode acessar esse conjunto resumido de princípios para escrever um bom código Python fornecendo `import this` ao seu interpretador. Não reproduzirei todo o "Zen de Python" aqui, mas compartilharei algumas linhas para ajudar a entender por que elas devem ser importantes para você como um programador Python iniciante.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

(**Bonito é melhor que feio**) Os programadores Python adotam a noção de que o código pode ser bonito e elegante. Em programação, as pessoas resolvem problemas. Os programadores sempre respeitaram soluções bem projetadas, eficientes e até mesmo bonitas para os problemas. À medida que conhecer mais a linguagem Python e usá-la para escrever mais códigos, uma pessoa poderá espiar por sobre seu ombro um dia e dizer: "Uau, que código bonito!".

```
Simple is better than complex.
```

(**Simples é melhor que complexo**) Se você puder escolher entre uma solução simples e outra complexa, e as duas funcionarem, utilize a solução simples. Seu código será mais fácil de manter, e será mais simples para você e para outras pessoas desenvolverem com base nesse código posteriormente.

```
Complex is better than complicated.
```

(**Complexo é melhor que complicado**) A vida real é confusa e, às vezes, uma solução simples para um problema não é viável. Nesse caso, utilize a solução mais simples possível que funcione.

```
Readability counts.
```

(**Legibilidade conta**) Mesmo quando seu código for complexo, procure deixá-lo legível. Quando trabalhar com um projeto que envolva códigos complexos, procure escrever comentários informativos para esse código.

```
There should be one-- and preferably only one --obvious way to do it.
```

(**Deve haver uma – e, de preferência, apenas uma – maneira óbvia de fazer algo**) Se for solicitado a dois programadores Python que resolvam o mesmo problema, eles deverão apresentar soluções razoavelmente compatíveis. Isso não quer dizer que não haja espaço para a criatividade em programação. Pelo contrário! No entanto, boa parte da programação consiste em usar abordagens pequenas e comuns para situações simples em um projeto maior e mais criativo. Os detalhes de funcionamento de seus programas devem fazer sentido para outros programadores Python.

```
Now is better than never.
```

(**Agora é melhor que nunca**) Você poderia passar o resto de sua vida conhecendo todas as complexidades de Python e da programação em geral, mas, nesse caso, jamais concluiria qualquer projeto. Não tente escrever um código perfeito; escreva um código que funcione e, então, decida se deve aperfeiçoá-lo nesse projeto ou passar para algo novo.

Ao prosseguir para o próximo capítulo e começar a explorar tópicos mais sofisticados, procure ter em mente essa filosofia de simplicidade e clareza. Programadores experientes respeitarão mais o seu código e ficarão felizes

em oferecer feedback e colaborar com você em projetos interessantes.

FAÇA VOCÊ MESMO

2.11 – Zen de Python: Digite `import this` em uma sessão de terminal de Python e dê uma olhada nos princípios adicionais.

Resumo

Neste capítulo aprendemos a trabalhar com variáveis. Aprendemos a usar nomes descritivos para as variáveis e a resolver erros de nomes e de sintaxe quando surgirem. Vimos o que são strings e como exibi-las usando letras minúsculas, letras maiúsculas e iniciais maiúsculas. No início, utilizamos espaços em branco para organizar a saída e aprendemos a remover caracteres em branco desnecessários de diferentes partes de uma string. Começamos a trabalhar com inteiros e números de ponto flutuante, e lemos a respeito de alguns comportamentos inesperados com os quais devemos tomar cuidado quando trabalharmos com dados numéricos. Também aprendemos a escrever comentários explicativos para que você e outras pessoas leiam seu código com mais facilidade. Por fim, lemos a respeito da filosofia de manter seu código o mais simples possível, sempre que puder.

No Capítulo 3 aprenderemos a armazenar coleções de informações em variáveis chamadas *listas*. Veremos como percorrer uma lista, manipulando qualquer informação que ela tiver.

3
INTRODUÇÃO ÀS LISTAS

Neste capítulo e no próximo, veremos o que são listas e como começar a trabalhar com os elementos de uma lista. As listas permitem armazenar conjuntos de informações em um só lugar, independentemente de termos alguns itens ou milhões deles. As listas são um dos recursos mais eficazes de Python, prontamente acessíveis aos novos programadores, e elas agregam muitos conceitos importantes em programação.

O que é uma lista?

Uma *lista* é uma coleção de itens em uma ordem em particular. Podemos criar uma lista que inclua as letras do alfabeto, os dígitos de 0 a 9 ou os nomes de todas as pessoas de sua família. Você pode colocar qualquer informação que quiser em uma lista, e os itens de sua lista não precisam estar relacionados de nenhum modo em particular. Como uma lista geralmente contém mais de um elemento, é uma boa ideia deixar seu nome no plural, por exemplo, `letters`, `digits` ou `names`.

Em Python, colchetes (`[]`) indicam uma lista, e elementos individuais da lista são separados por vírgulas. Eis um exemplo simples de uma lista que contém alguns tipos de bicicleta:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

Se você pedir a Python que exiba uma lista, ela devolverá a representação da lista, incluindo os colchetes:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Como essa não é a saída que você quer que seus usuários vejam, vamos

aprender a acessar os elementos individuais de uma lista.

Acessando elementos de uma lista

Listas são coleções ordenadas, portanto você pode acessar qualquer elemento de uma lista informando a posição – ou *índice* – do item desejado ao interpretador Python. Para acessar um elemento de uma lista, escreva o nome da lista seguido do índice do item entre colchetes.

Por exemplo, vamos extrair a primeira bicicleta da lista `bicycles`:

```
  bicycles = ['trek', 'cannondale', 'redline', 'specialized']
u print(bicycles[0])
```

A sintaxe para isso está em u. Quando solicitamos um item simples de uma lista, Python devolve apenas esse elemento, sem colchetes ou aspas:

```
trek
```

Esse é o resultado que você quer que seus usuários vejam – uma saída limpa, formatada de modo elegante.

Também podemos usar os métodos de string do Capítulo 2 em qualquer elemento de uma lista. Por exemplo, podemos formatar o elemento `'trek'` de modo mais bonito usando o método `title()`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Esse exemplo gera a mesma saída do exemplo anterior, exceto pelo fato de `'Trek'` começar com uma letra maiúscula.

A posição dos índices começa em 0, e não em 1

Python considera que o primeiro item de uma lista está na posição 0, e não na posição 1. Isso é válido para a maioria das linguagens de programação, e o motivo tem a ver com o modo como as operações em lista são implementadas em um nível mais baixo. Se estiver recebendo resultados inesperados, verifique se você não está cometendo um erro simples de deslocamento de um.

O segundo item de uma lista tem índice igual a 1. Usando esse sistema

simples de contagem, podemos obter qualquer elemento que quisermos de uma lista subtraindo um de sua posição na lista. Por exemplo, para acessar o quarto item de uma lista, solicite o item no índice 3.

As instruções a seguir acessam as bicicletas nos índices `1` e `3`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

Esse código devolve a segunda e a quarta bicicletas da lista:

```
cannondale
specialized
```

Python tem uma sintaxe especial para acessar o último elemento de uma lista. Ao solicitar o item no índice `-1`, Python sempre devolve o último item da lista:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Esse código devolve o valor `'specialized'`. Essa sintaxe é bem útil, pois, com frequência, você vai querer acessar os últimos itens de uma lista sem saber exatamente o tamanho dela. Essa convenção também se estende a outros valores negativos de índice. O índice `-2` devolve o segundo item a partir do final da lista, o índice `-3` devolve o terceiro item a partir do final, e assim sucessivamente.

Usando valores individuais de uma lista

Você pode usar valores individuais de uma lista, exatamente como faria com qualquer outra variável. Por exemplo, podemos usar concatenação para criar uma mensagem com base em um valor de uma lista.

Vamos tentar obter a primeira bicicleta da lista e compor uma mensagem usando esse valor.

```
  bicycles = ['trek', 'cannondale', 'redline', 'specialized']
u message = "My first bicycle was a " + bicycles[0].title() + "."

  print(message)
```

Em u, compomos uma frase usando o valor em `bicycles[0]` e a armazenamos na variável `message`. A saída é uma frase simples sobre a primeira bicicleta da lista:

```
My first bicycle was a Trek.
```

FAÇA VOCÊ MESMO

Experimente criar estes programas pequenos para ter um pouco de experiência própria com listas em Python. Você pode criar uma nova pasta para os exercícios de cada capítulo a fim de mantê-los organizados.

3.1 – Nomes: Armazene os nomes de alguns de seus amigos em uma lista chamada `names`. Exiba o nome de cada pessoa acessando cada elemento da lista, um de cada vez.

3.2 – Saudações: Comece com a lista usada no Exercício 3.1, mas em vez de simplesmente exibir o nome de cada pessoa, apresente uma mensagem a elas. O texto de cada mensagem deve ser o mesmo, porém cada mensagem deve estar personalizada com o nome da pessoa.

3.3 – Sua própria lista: Pense em seu meio de transporte preferido, como motocicleta ou carro, e crie uma lista que armazene vários exemplos. Utilize sua lista para exibir uma série de frases sobre esses itens, como "Gostaria de ter uma moto Honda".

Alterando, acrescentando e removendo elementos

A maioria das listas que você criar será dinâmica, o que significa que você criará uma lista e então adicionará e removerá elementos dela à medida que seu programa executar. Por exemplo, você poderia criar um jogo em que um jogador atire em alienígenas que caem do céu. Poderia armazenar o conjunto inicial de alienígenas em uma lista e então remover um item da lista sempre que um alienígena for atingido. Sempre que um novo alienígena aparecer na tela, adicione-o à lista. Sua lista de alienígenas diminuirá e aumentará de tamanho no decorrer do jogo.

Modificando elementos de uma lista

A sintaxe para modificar um elemento é semelhante à sintaxe para acessar um elemento de uma lista. Para alterar um elemento, use o nome da lista

seguido do índice do elemento que você quer modificar e, então, forneça o novo valor que você quer que esse item tenha.

Por exemplo, vamos supor que temos uma lista de motocicletas, e que o primeiro item da lista seja `'honda'`. Como mudaríamos o valor desse primeiro item?

motorcycles.py

```
u motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

v motorcycles[0] = 'ducati'
  print(motorcycles)
```

O código em u define a lista original, com `'honda'` como o primeiro elemento. O código em v altera o valor do primeiro item para `'ducati'`. A saída mostra que o primeiro item realmente foi modificado e o restante da lista permaneceu igual:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

Você pode mudar o valor de qualquer item de uma lista, e não apenas o primeiro.

Acrescentando elementos em uma lista

Você pode acrescentar um novo elemento em uma lista por diversos motivos. Por exemplo, talvez você queira que novos alienígenas apareçam no jogo, pode querer acrescentar novos dados em uma visualização ou adicionar novos usuários registrados em um site que você criou. Python oferece várias maneiras de acrescentar novos dados em listas existentes.

Concatenando elementos no final de uma lista

A maneira mais simples de acrescentar um novo elemento em uma lista é *concatenar* o item na lista. Quando concatenamos um item em uma lista, o novo elemento é acrescentado no final. Usando a mesma lista que tínhamos no exemplo anterior, adicionaremos o novo elemento `'ducati'`

no final da lista:

```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

u motorcycles.append('ducati')
  print(motorcycles)
```

O método `append()` em **u** acrescenta `'ducati'` no final da lista sem afetar qualquer outro elemento:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

O método `append()` facilita criar listas dinamicamente. Por exemplo, podemos começar com uma lista vazia e então acrescentar itens à lista usando uma série de instruções `append()`. Usando uma lista vazia, vamos adicionar os elementos `'honda'`, `'yamaha'` e `'suzuki'` à lista:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

A lista resultante tem exatamente o mesmo aspecto da lista dos exemplos anteriores:

```
['honda', 'yamaha', 'suzuki']
```

Criar listas dessa maneira é bem comum, pois, com frequência, você não conhecerá os dados que seus usuários querem armazenar em um programa até que ele esteja executando. Para deixar que seus usuários tenham o controle, comece definindo uma lista vazia que armazenará os valores dos usuários. Em seguida, concatene cada novo valor fornecido à lista que você acabou de criar.

Inserindo elementos em uma lista

Você pode adicionar um novo elemento em qualquer posição de sua lista

usando o método `insert()`. Faça isso especificando o índice do novo elemento e o valor do novo item.

```
  motorcycles = ['honda', 'yamaha', 'suzuki']

u motorcycles.insert(0, 'ducati')
  print(motorcycles)
```

Nesse exemplo, o código em u insere o valor `'ducati'` no início da lista. O método `insert()` abre um espaço na posição `0` e armazena o valor `'ducati'` nesse local. Essa operação desloca todos os demais valores da lista uma posição à direita:

```
  ['ducati', 'honda', 'yamaha', 'suzuki']
```

Removendo elementos de uma lista

Com frequência, você vai querer remover um item ou um conjunto de itens de uma lista. Por exemplo, quando um jogador atinge um alienígena no céu com um tiro, é bem provável que você vá querer removê-lo da lista de alienígenas ativos. Se um usuário decidir cancelar a conta em sua aplicação web, você vai querer remover esse usuário da lista de usuários ativos. Você pode remover um item de acordo com sua posição na lista ou seu valor.

Removendo um item usando a instrução del

Se a posição do item que você quer remover de uma lista for conhecida, a instrução `del` poderá ser usada.

```
  motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

u del motorcycles[0]
  print(motorcycles)
```

O código em u usa `del` para remover o primeiro item, `'honda'`, da lista de motocicletas:

```
  ['honda', 'yamaha', 'suzuki']
  ['yamaha', 'suzuki']
```

Você pode remover um item de qualquer posição em uma lista usando a instrução `del`, se souber qual é o seu índice. Por exemplo, eis o modo de remover o segundo item, `'yamaha'`, da lista:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

A segunda motocicleta é apagada da lista:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

Nos dois exemplos não podemos mais acessar o valor que foi removido da lista após a instrução `del` ter sido usada.

Removendo um item com o método pop()

Às vezes, você vai querer usar o valor de um item depois de removê-lo de uma lista. Por exemplo, talvez você queira obter as posições *x* e *y* de um alienígena que acabou de ser atingido para que possa desenhar uma explosão nessa posição. Em uma aplicação web, você poderia remover um usuário de uma lista de membros ativos e então adicioná-lo a uma lista de membros inativos.

O método `pop()` remove o último item de uma lista, mas permite que você trabalhe com esse item depois da remoção. O termo *pop* deriva de pensar em uma lista como se fosse uma pilha de itens e remover um item (fazer um pop) do topo da pilha. Nessa analogia, o topo da pilha corresponde ao final da lista.

Vamos fazer um pop de uma motocicleta da lista de motocicletas:

```
u motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

v popped_motorcycle = motorcycles.pop()
w print(motorcycles)
x print(popped_motorcycle)
```

Começamos definindo e exibindo a lista motorcycles em u. Em v fazemos pop de um valor da lista e o armazenamos na variável popped_motorcycle. Exibimos a lista em w para mostrar que um valor foi removido da lista. Então exibimos o valor removido em x para provar que ainda temos acesso ao valor removido.

A saída mostra que o valor 'suzuki' foi removido do final da lista e agora está armazenado na variável popped_motorcycle:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

Como esse método pop() poderia ser útil? Suponha que as motocicletas da lista estejam armazenadas em ordem cronológica, de acordo com a época em que fomos seus proprietários. Se for esse o caso, podemos usar o método pop() para exibir uma afirmação sobre a última motocicleta que compramos:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print("The last motorcycle I owned was a " + last_owned.title() + ".")
```

A saída é uma frase simples sobre a motocicleta mais recente que tivemos:

```
The last motorcycle I owned was a Suzuki.
```

Removendo itens de qualquer posição em uma lista

Na verdade, você pode usar pop() para remover um item de qualquer posição em uma lista se incluir o índice do item que você deseja remover entre parênteses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

u first_owned = motorcycles.pop(0)
v print('The first motorcycle I owned was a ' + first_owned.title() +
'.')
```

Começamos fazendo pop da primeira motocicleta da lista em u e, então,

exibimos uma mensagem sobre essa motocicleta em v. A saída é uma frase simples que descreve a primeira motocicleta que eu tive:

```
The first motorcycle I owned was a Honda.
```

Lembre-se de que, sempre que usar `pop()`, o item com o qual você trabalhar não estará mais armazenado na lista.

Se você não tiver certeza se deve usar a instrução `del` ou o método `pop()`, eis um modo fácil de decidir: quando quiser apagar um item de uma lista e esse item não vai ser usado de modo algum, utilize a instrução `del`; se quiser usar um item à medida que removê-lo, utilize o método `pop()`.

Removendo um item de acordo com o valor

Às vezes, você são saberá a posição do valor que quer remover de uma lista. Se conhecer apenas o valor do item que deseja remover, o método `remove()` poderá ser usado.

Por exemplo, vamos supor que queremos remover o valor `'ducati'` da lista de motocicletas.

```
 motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
 print(motorcycles)

u motorcycles.remove('ducati')
 print(motorcycles)
```

O código em u diz a Python para descobrir em que lugar `'ducati'` aparece na lista e remover esse elemento:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

Também podemos usar o método `remove()` para trabalhar com um valor que está sendo removido de uma lista. Vamos remover o valor `'ducati'` e exibir um motivo para removê-lo da lista:

```
u motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
 print(motorcycles)

v too_expensive = 'ducati'
```

```
w motorcycles.remove(too_expensive)
  print(motorcycles)
x print("\nA " + too_expensive.title() + " is too expensive for me.")
```

Após definir a lista em u, armazenamos o valor `'ducati'` em uma variável chamada `too_expensive` v. Então utilizamos essa variável para dizer a Python qual valor deve ser removido da lista em w. Em x, o valor `'ducati'` foi removido da lista, mas continua armazenado na variável `too_expensive`, permitindo exibir uma frase sobre o motivo pelo qual removemos `'ducati'` da lista de motocicletas:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

NOTA O método `remove()` apaga apenas a primeira ocorrência do valor que você especificar. Se houver a possibilidade de o valor aparecer mais de uma vez na lista, será necessário usar um laço para determinar se todas as ocorrências desse valor foram removidas. Aprenderemos a fazer isso no Capítulo 7.

FAÇA VOCÊ MESMO

Os exercícios a seguir são um pouco mais complexos que aqueles do Capítulo 2, porém darão a você a oportunidade de usar listas de todas as formas descritas.

3.4 – Lista de convidados: Se pudesse convidar alguém, vivo ou morto, para o jantar, quem você convidaria? Crie uma lista que inclua pelo menos três pessoas que você gostaria de convidar para jantar. Em seguida, utilize sua lista para exibir uma mensagem para cada pessoa, convidando-a para jantar.

3.5 – Alterando a lista de convidados: Você acabou de saber que um de seus convidados não poderá comparecer ao jantar, portanto será necessário enviar um novo conjunto de convites. Você deverá pensar em outra pessoa para convidar.

- Comece com seu programa do Exercício 3.4. Acrescente uma instrução `print` no final de seu programa, especificando o nome do convidado que não poderá comparecer.
- Modifique sua lista, substituindo o nome do convidado que não poderá comparecer pelo nome da nova pessoa que você está convidando.
- Exiba um segundo conjunto de mensagens com o convite, uma para cada pessoa que continua presente em sua lista.

3.6 – Mais convidados: Você acabou de encontrar uma mesa de jantar maior,

portanto agora tem mais espaço disponível. Pense em mais três convidados para o jantar.

- Comece com seu programa do Exercício 3.4 ou do Exercício 3.5. Acrescente uma instrução `print` no final de seu programa informando às pessoas que você encontrou uma mesa de jantar maior.
- Utilize `insert()` para adicionar um novo convidado no início de sua lista.
- Utilize `insert()` para adicionar um novo convidado no meio de sua lista.
- Utilize `append()` para adicionar um novo convidado no final de sua lista.
- Exiba um novo conjunto de mensagens de convite, uma para cada pessoa que está em sua lista.

3.7 – Reduzindo a lista de convidados: Você acabou de descobrir que sua nova mesa de jantar não chegará a tempo para o jantar e tem espaço para somente dois convidados.

- Comece com seu programa do Exercício 3.6. Acrescente uma nova linha que mostre uma mensagem informando que você pode convidar apenas duas pessoas para o jantar.
- Utilize `pop()` para remover os convidados de sua lista, um de cada vez, até que apenas dois nomes permaneçam em sua lista. Sempre que remover um nome de sua lista, mostre uma mensagem a essa pessoa, permitindo que ela saiba que você sente muito por não poder convidá-la para o jantar.
- Apresente uma mensagem para cada uma das duas pessoas que continuam na lista, permitindo que elas saibam que ainda estão convidadas.
- Utilize `del` para remover os dois últimos nomes de sua lista, de modo que você tenha uma lista vazia. Mostre sua lista para garantir que você realmente tem uma lista vazia no final de seu programa.

Organizando uma lista

Com frequência, suas listas serão criadas em uma ordem imprevisível, pois nem sempre você poderá controlar a ordem em que seus usuários fornecem seus dados. Embora isso seja inevitável na maioria das circunstâncias, com frequência você vai querer apresentar suas informações em uma ordem em particular. Às vezes, você vai querer preservar a ordem original de sua lista, enquanto, em outras ocasiões, vai querer alterar essa ordem. Python oferece várias maneiras de organizar suas listas de acordo com a situação.

Ordenando uma lista de forma permanente com o método sort()

O método `sort()` de Python faz com que seja relativamente fácil ordenar uma lista. Suponha que temos uma lista de carros e queremos alterar a ordem da lista para armazenar os itens em ordem alfabética. Para simplificar essa tarefa, vamos supor que todos os valores da lista usam letras minúsculas.

cars.py

```
  cars = ['bmw', 'audi', 'toyota', 'subaru']
u cars.sort()
  print(cars)
```

O método `sort()` mostrado em u altera a ordem da lista de forma permanente. Os carros agora estão em ordem alfabética e não podemos mais retornar à ordem original.

```
['audi', 'bmw', 'subaru', 'toyota']
```

Também podemos ordenar essa lista em ordem alfabética inversa, passando o argumento `reverse=True` ao método `sort()`. O exemplo a seguir ordena a lista de carros em ordem alfabética inversa:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Novamente, a ordem da lista foi permanentemente alterada:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Ordenando uma lista temporariamente com a função sorted()

Para preservar a ordem original de uma lista, mas apresentá-la de forma ordenada, podemos usar a função `sorted()`. A função `sorted()` permite exibir sua lista em uma ordem em particular, mas não afeta a ordem propriamente dita da lista.

Vamos testar essa função na lista de carros.

```
  cars = ['bmw', 'audi', 'toyota', 'subaru']

u print("Here is the original list:")
```

```
  print(cars)

v print("\nHere is the sorted list:")
  print(sorted(cars))

w print("\nHere is the original list again:")
  print(cars)
```

Inicialmente, exibimos a lista em sua ordem original em u e depois, em ordem alfabética em v. Depois que a lista é exibida na nova ordem, mostramos que a lista continua armazenada em sua ordem original em w.

```
  Here is the original list:
  ['bmw', 'audi', 'toyota', 'subaru']

  Here is the sorted list:
  ['audi', 'bmw', 'subaru', 'toyota']

x Here is the original list again:
  ['bmw', 'audi', 'toyota', 'subaru']
```

Observe que a lista preserva sua ordem original em x, depois que a função `sorted()` foi usada. Essa função também pode aceitar um argumento `reverse=True` se você quiser exibir uma lista em ordem alfabética inversa.

NOTA Ordenar uma lista em ordem alfabética é um pouco mais complicado quando todos os valores não utilizam letras minúsculas. Há várias maneiras de interpretar letras maiúsculas quando decidimos por uma sequência de ordenação, e especificar a ordem exata pode apresentar um nível de complexidade maior que aquele com que queremos lidar no momento. No entanto, a maior parte das abordagens à ordenação terá diretamente como base o que aprendemos nesta seção.

Exibindo uma lista em ordem inversa

Para inverter a ordem original de uma lista, podemos usar o método `reverse()`. Se armazenarmos originalmente a lista de carros em ordem cronológica, de acordo com a época em que fomos seus proprietários, poderemos facilmente reorganizar a lista em ordem cronológica inversa:

```
  cars = ['bmw', 'audi', 'toyota', 'subaru']
  print(cars)
```

```
cars.reverse()
print(cars)
```

Observe que `reverse()` não reorganiza em ordem alfabética inversa; ele simplesmente inverte a ordem da lista:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

O método `reverse()` muda a ordem de uma lista de forma permanente, mas podemos restaurar a ordem original a qualquer momento aplicando `reverse()` à mesma lista uma segunda vez.

Descobrindo o tamanho de uma lista

Podemos rapidamente descobrir o tamanho de uma lista usando a função `len()`. A lista no exemplo a seguir tem quatro itens, portanto seu tamanho é `4`:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

Você achará `len()` útil quando precisar identificar o número de alienígenas que ainda precisam ser atingidos em um jogo, determinar a quantidade de dados que você precisa administrar em uma visualização ou descobrir o número de usuários registrados em um site, entre outras tarefas.

NOTA Python conta os itens de uma lista começando em um, portanto você não deverá se deparar com nenhum erro de deslocamento de um ao determinar o tamanho de uma lista.

FAÇA VOCÊ MESMO

3.8 – Conhecendo o mundo: Pense em pelo menos cinco lugares do mundo que você gostaria de visitar.

- Armazene as localidades em uma lista. Certifique-se de que a lista não esteja em ordem alfabética.

- Exiba sua lista na ordem original. Não se preocupe em exibir a lista de forma elegante; basta exibi-la como uma lista Python pura.
- Utilize `sorted()` para exibir sua lista em ordem alfabética, sem modificar a lista propriamente dita.
- Mostre que sua lista manteve sua ordem original exibindo-a.
- Utilize `sorted()` para exibir sua lista em ordem alfabética inversa sem alterar a ordem da lista original.
- Mostre que sua lista manteve sua ordem original exibindo-a novamente.
- Utilize `reverse()` para mudar a ordem de sua lista. Exiba a lista para mostrar que sua ordem mudou.
- Utilize `reverse()` para mudar a ordem de sua lista novamente. Exiba a lista para mostrar que ela voltou à sua ordem original.
- Utilize `sort()` para mudar sua lista de modo que ela seja armazenada em ordem alfabética. Exiba a lista para mostrar que sua ordem mudou.
- Utilize `sort()` para mudar sua lista de modo que ela seja armazenada em ordem alfabética inversa. Exiba a lista para mostrar que sua ordem mudou.

3.9 – Convidados para o jantar: Trabalhando com um dos programas dos Exercícios de 3.4 a 3.7 (páginas 80 e 81), use `len()` para exibir uma mensagem informando o número de pessoas que você está convidando para o jantar.

3.10 – Todas as funções: Pensa em algo que você poderia armazenar em uma lista. Por exemplo, você poderia criar uma lista de montanhas, rios, países, cidades, idiomas ou qualquer outro item que quiser. Escreva um programa que crie uma lista contendo esses itens e então utilize cada função apresentada neste capítulo pelo menos uma vez.

Evitando erros de índice quando trabalhar com listas

Um tipo de erro é comum quando trabalhamos com listas pela primeira vez. Suponha que temos uma lista com três itens e você solicite o quarto item:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

Esse exemplo resulta em um *erro de índice*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[3])
```

```
IndexError: list index out of range
```

Python tenta fornecer o item no índice 3. Porém, quando pesquisa a lista, nenhum item de `motorcycles` tem índice igual a 3. Por causa da natureza deslocada de um na indexação de listas, esse erro é característico. As pessoas acham que o terceiro item é o item de número 3, pois começam a contar a partir de 1. Contudo, em Python, o terceiro item é o de número 2, pois a indexação começa em 0.

Um erro de índice quer dizer que Python não é capaz de determinar o índice solicitado. Se um erro de índice ocorrer em seu programa, tente ajustar o índice que você está solicitando em um. Então execute o programa novamente para ver se os resultados estão corretos.

Tenha em mente que, sempre que quiser acessar o último item de uma lista, você deve usar o índice `-1`. Isso sempre funcionará, mesmo que sua lista tenha mudado de tamanho desde a última vez que você a acessou:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

O índice `-1` sempre devolve o último item de uma lista – nesse caso, é o valor `'suzuki'`:

```
'suzuki'
```

A única ocasião em que essa abordagem causará um erro é quando solicitamos o último item de uma lista vazia:

```
motorcycles = []
print(motorcycles[-1])
```

Não há nenhum item em `motorcycles`, portanto Python devolve outro erro de índice:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

NOTA Se um erro de índice ocorrer e você não consegue descobrir como resolvê-lo, experimente exibir sua lista ou simplesmente mostrar o tamanho dela. Sua lista

poderá estar bem diferente do que você imaginou, em especial se ela foi tratada dinamicamente pelo seu programa. Ver a lista propriamente dita ou o número exato de itens que ela contém pode ajudar a entender esses erros de lógica.

FAÇA VOCÊ MESMO

3.11 – Erro proposital: Se você ainda não recebeu um erro de índice em um de seus programas, tente fazer um erro desse tipo acontecer. Altere um índice em um de seus programas de modo a gerar um erro de índice. Não se esqueça de corrigir o erro antes de fechar o programa.

Resumo

Neste capítulo conhecemos as listas e vimos como trabalhar com os itens individuais de uma lista. Aprendemos a definir uma lista e a adicionar e remover elementos. Vimos como ordenar as listas de modo permanente e temporário para fins de exibição. Também vimos como descobrir o tamanho de uma lista e aprendemos a evitar erros de índice quando trabalhamos com listas.

No Capítulo 4 veremos como trabalhar com itens de uma lista de maneira mais eficiente. Ao percorrer todos os itens em um laço usando apenas algumas linhas de código, poderemos trabalhar de modo eficiente, mesmo quando a lista contiver milhares ou milhões de itens.

4
TRABALHANDO COM LISTAS

No Capítulo 3 aprendemos a criar uma lista simples e a trabalhar com os elementos individuais de uma lista. Neste capítulo veremos como percorrer uma lista inteira com um *laço* usando apenas algumas linhas de código, independentemente do tamanho da lista. Percorrer listas com laços permite executar a mesma ação – ou conjunto de ações – em todos os itens de uma lista. Como resultado, você poderá trabalhar de modo eficiente com listas de qualquer tamanho, incluindo aquelas com milhares ou até mesmo milhões de itens.

Percorrendo uma lista inteira com um laço

Com frequência, você vai querer percorrer todas as entradas de uma lista, executando a mesma tarefa em cada item. Por exemplo, em um jogo, você pode mover todos os elementos da tela de acordo com a mesma distância; em uma lista de números, talvez você queira executar a mesma operação estatística em todos os elementos. Quem sabe você queira exibir cada um dos títulos de uma lista de artigos em um site. Quando quiser executar a mesma ação em todos os itens de uma lista, você pode usar o laço `for` de Python.

Vamos supor que temos uma lista de nomes de mágicos e queremos exibir todos os nomes da lista. Poderíamos fazer isso recuperando cada nome da lista individualmente, mas essa abordagem poderia causar vários problemas. Para começar, seria repetitivo fazer isso com uma lista longa de nomes. Além disso, teríamos que alterar o nosso código sempre que o tamanho da lista mudasse. Um laço `for` evita esses dois problemas ao permitir que Python administre essas questões internamente.

Vamos usar um laço `for` para exibir cada um dos nomes de uma lista de mágicos:

magicians.py

```
u magicians = ['alice', 'david', 'carolina']
v for magician in magicians:
w     print(magician)
```

Começamos definindo uma lista em u, exatamente como fizemos no Capítulo 3. Em v definimos um laço `for`. Essa linha diz a Python para extrair um nome da lista `magicians` e armazená-lo na variável `magician`. Em w dizemos a Python para exibir o nome que acabou de ser armazenado em `magician`. O interpretador então repete as linhas v e w, uma vez para cada nome da lista. Ler esse código como "para todo mágico na lista de mágicos, exiba o nome do mágico" pode ajudar. A saída é uma exibição simples de cada nome da lista:

```
alice
david
carolina
```

Observando os laços com mais detalhes

O conceito de laços é importante porque é uma das maneiras mais comuns para um computador automatizar tarefas repetitivas. Por exemplo, em um laço simples como o que usamos em *magicians.py*, Python inicialmente lê a primeira linha do laço:

```
for magician in magicians:
```

Essa linha diz a Python para extrair o primeiro valor da lista `magicians` e armazená-lo na variável `magician`. O primeiro valor é `'alice'`. O interpretador então lê a próxima linha:

```
    print(magician)
```

Python exibe o valor atual de `magician`, que ainda é `'alice'`. Como a lista contém mais valores, o interpretador retorna à primeira linha do laço:

```
for magician in magicians:
```

Python recupera o próximo nome da lista, que é `'david'`, e armazena esse valor em `magician`. Então ele executa a linha:

```
    print(magician)
```

Python exibe o valor atual de `magician`, que agora é `'david'`, novamente. O interpretador repete todo o laço mais uma vez com o último valor da lista, que é `'carolina'`. Como não há mais valores na lista, Python passa para a próxima linha do programa. Nesse caso, não há mais nada depois do laço `for`, portanto o programa simplesmente termina.

Quando usar laços pela primeira vez, tenha em mente que o conjunto de passos será repetido, uma vez para cada item da lista, não importa quantos itens haja na lista. Se você tiver um milhão de itens em sua lista, Python repetirá esses passos um milhão de vezes – e geralmente o fará bem rápido.

Tenha em mente também que quando escrever seus próprios laços `for`, você poderá escolher qualquer nome que quiser para a variável temporária que armazena cada valor da lista. No entanto, é conveniente escolher um nome significativo, que represente um único item da lista. Por exemplo, eis uma boa maneira de iniciar um laço `for` para uma lista de gatos, uma lista de cachorros e uma lista genérica de itens:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

Essas convenções de nomenclatura podem ajudar você a acompanhar a ação executada em cada item em um laço `for`. O uso de nomes no singular e no plural pode ajudar a identificar se a seção de código atua em um único elemento da lista ou em toda a lista.

Executando mais tarefas em um laço for

Você pode fazer praticamente de tudo com cada item em um laço `for`. Vamos expandir o exemplo anterior exibindo uma mensagem a cada mágico, informando-lhes que realizaram um ótimo truque:

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
u     print(magician.title() + ", that was a great trick!")
```

A única diferença nesse código está em u, em que compomos uma

mensagem para cada mágico, começando com o nome desse mágico. Na primeira vez em que passamos pelo laço, o valor do mágico é `'alice'`, portanto Python inicia a primeira mensagem com o nome `'Alice'`. Na segunda vez que compormos a mensagem, ela começará com `'David'` e na terceira vez, a mensagem começará com `'Carolina'`.

A saída mostra uma mensagem personalizada para cada mágico da lista:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

Também podemos escrever tantas linhas de código quantas quisermos no laço `for`. Considera-se que toda linha indentada após a linha `for magician in magicians` está *dentro do laço*, e cada linha indentada é executada uma vez para cada valor da lista. Assim, você pode executar o volume de trabalho que quiser com cada valor da lista.

Vamos acrescentar uma segunda linha em nossa mensagem, informando a cada mágico que estamos ansiosos para ver o seu próximo truque:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
u     print("I can't wait to see your next trick, " + magician.title() +
".\n")
```

Como indentamos as duas instruções `print`, cada linha será executada uma vez para cada mágico da lista. A quebra de linha (`"\n"`) na segunda instrução `print` **u** insere uma linha em branco após cada passagem pelo laço. Com isso, criamos um conjunto de mensagens agrupadas de forma organizada para cada pessoa na lista:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Podemos usar quantas linhas quisermos em nossos laços `for`. Na prática, muitas vezes você achará útil efetuar várias operações com cada item de uma lista quando usar um laço `for`.

Fazendo algo após um laço for

O que acontece quando um laço `for` acaba de executar? Geralmente você vai querer fazer uma síntese de um bloco de saída ou passar para outra atividade que seu programa deva executar.

Qualquer linha de código após o laço `for` que não estiver indentada será executada uma vez, sem repetição. Vamos escrever um agradecimento ao grupo de mágicos como um todo, agradecendo-lhes por apresentar um show excelente. Para exibir essa mensagem ao grupo após todas as mensagens individuais terem sido apresentadas, colocamos a mensagem de agradecimento depois do laço `for`, sem indentação:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() +
".\n")

u print("Thank you, everyone. That was a great magic show!")
```

As duas primeiras instruções `print` são repetidas uma vez para cada mágico da lista, como vimos antes. No entanto, como a linha em u não está indentada, ela será exibida apenas uma vez:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

Quando processar dados com um laço `for`, você verá que essa é uma boa

maneira de sintetizar uma operação realizada em todo um conjunto de dados. Por exemplo, um laço `for` pode ser usado para inicializar um jogo percorrendo uma lista de personagens e exibindo cada um deles na tela. Você pode então escrever um bloco não indentado após esse laço, que exiba um botão Play Now (Jogue agora) depois que todos os personagens tiverem sido desenhados na tela.

Evitando erros de indentação

Python usa indentação para determinar se uma linha de código está conectada à linha antes dela. Nos exemplos anteriores, as linhas que exibiam mensagens aos mágicos individuais faziam parte do laço `for` porque estavam indentadas. O uso de indentação por Python deixa o código bem fácil de ler. Basicamente, Python usa espaços em branco para forçar você a escrever um código formatado de modo organizado, com uma estrutura visual clara. Em programas Python mais longos, você perceberá que há blocos de código indentados em alguns níveis diferentes. Esses níveis de indentação ajudam a ter uma noção geral da organização do programa como um todo.

Quando começar a escrever código que dependa de uma indentação apropriada, você deverá tomar cuidado com alguns *erros comuns de indentação*. Por exemplo, às vezes, as pessoas indentam blocos de código que não precisam estar indentados ou se esquecem de indentar blocos que deveriam estar indentados. Ver exemplos desses erros agora ajudará a evitá-los no futuro e a corrigi-los quando aparecerem em seus próprios programas.

Vamos analisar alguns dos erros mais comuns de indentação.

Esquecendo-se de indentar

Sempre indente a linha após a instrução `for` em um laço. Se você se esquecer, Python o avisará:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
```

```
  for magician in magicians:
u print(magician)
```

A instrução `print` em u deveria estar indentada, mas não está. Quando Python espera um bloco indentado e não encontra um, ele mostra a linha em que o problema ocorreu.

```
    File "magicians.py", line 3
      print(magician)
          ^
  IndentationError: expected an indented block
```

Geralmente podemos resolver esse tipo de erro indentando a linha ou as linhas logo depois da instrução `for`.

Esquecendo-se de indentar linhas adicionais

Às vezes, seu laço executará sem erros, mas não produzirá o resultado esperado. Isso pode acontecer quando você tenta realizar várias tarefas em um laço e se esquece de indentar algumas de suas linhas.

Por exemplo, eis o que acontece quando nos esquecemos de indentar a segunda linha do laço que diz a cada mágico que estamos ansiosos para ver o seu próximo truque:

```
  magicians = ['alice', 'david', 'carolina']
  for magician in magicians:
      print(magician.title() + ", that was a great trick!")
u print("I can't wait to see your next trick, " + magician.title() +
".\n")
```

A instrução `print` em u deveria estar indentada, mas como Python encontra pelo menos uma linha indentada após a instrução `for`, um erro não é informado. Como resultado, a primeira instrução `print` é executada uma vez para cada nome da lista, pois está indentada. A segunda instrução `print` não está indentada, portanto é executada somente uma vez, depois que o laço terminar de executar. Como o valor final de `magician` é `'carolina'`, ela é a única que recebe a mensagem de “ansiosos pelo próximo truque”:

```
  Alice, that was a great trick!
```

```
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Esse é um *erro de lógica*. É um código Python com sintaxe válida, mas ele não gera o resultado desejado, pois há um problema em sua lógica. Se você espera ver uma determinada ação ser repetida uma vez para cada item de uma lista, mas ela é executada apenas uma vez, verifique se não é preciso simplesmente indentar uma linha ou um grupo de linhas.

Indentando desnecessariamente

Se você, acidentalmente, indentar uma linha que não precisa ser indetada, Python o informará a respeito da indentação inesperada:

hello_world.py

```
  message = "Hello Python world!"
u     print(message)
```

Não precisamos indentar a instrução `print` em u, pois ela não *pertence* à linha antes dela; assim, Python informa esse erro:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Você pode evitar erros inesperados de indentação ao indentar apenas quando houver um motivo específico para isso. Nos programas que estamos escrevendo no momento, as únicas linhas que precisam ser indentadas são as ações que queremos repetir para cada item em um laço `for`.

Indentando desnecessariamente após o laço

Se você acidentalmente indentar um código que deva executar após um laço ter sido concluído, esse código será repetido uma vez para cada item da lista. Às vezes, isso faz Python informar um erro, mas, geralmente, você terá um erro de lógica simples.

Por exemplo, vamos ver o que acontece quando indentamos por acidente a linha que agradece aos mágicos como um grupo por apresentarem um bom show:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() +
".\n")

u     print("Thank you everyone, that was a great magic show!")
```

Pelo fato de a linha em u estar indentada, ela é exibida uma vez para cada pessoa da lista, como podemos ver em v:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

v Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

v Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

v Thank you everyone, that was a great magic show!
```

Há outro erro de lógica, semelhante àquele da seção "Esquecendo-se de indentar linhas adicionais". Como Python não sabe o que você está querendo fazer com seu código, ele executará todo código que estiver escrito com uma sintaxe válida. Se uma ação for repetida muitas vezes quando deveria ser executada apenas uma vez, verifique se você não precisa simplesmente deixar de indentar o código dessa ação.

Esquecendo os dois-pontos

Os dois-pontos no final de uma instrução `for` diz a Python para interpretar a próxima linha como o início de um laço.

```
magicians = ['alice', 'david', 'carolina']
```

```
u for magician in magicians
     print(magician)
```

Se você se esquecer acidentalmente de colocar os dois-pontos, como em u, você terá um erro de sintaxe, pois Python não sabe o que você está tentando fazer. Embora esse seja um erro fácil de corrigir, nem sempre é um erro fácil de identificar. Você ficaria surpreso com a quantidade de tempo gasto pelos programadores à procura de erros de um único caractere como esse. Erros desse tipo são difíceis de encontrar, pois, com frequência, vemos somente aquilo que esperamos ver.

FAÇA VOCÊ MESMO

4.1 – Pizzas: Pense em pelo menos três tipos de pizzas favoritas. Armazene os nomes dessas pizzas e, então, utilize um laço `for` para exibir o nome de cada pizza.

- Modifique seu laço `for` para mostrar uma frase usando o nome da pizza em vez de exibir apenas o nome dela. Para cada pizza, você deve ter uma linha na saída contendo uma frase simples como *Gosto de pizza de pepperoni*.
- Acrescente uma linha no final de seu programa, fora do laço `for`, que informe quanto você gosta de pizza. A saída deve ser constituída de três ou mais linhas sobre os tipos de pizza que você gosta e de uma frase adicional, por exemplo, *Eu realmente adoro pizza!*

4.2 – Animais: Pense em pelo menos três animais diferentes que tenham uma característica em comum. Armazene os nomes desses animais em uma lista e, então, utilize um laço `for` para exibir o nome de cada animal.

- Modifique seu programa para exibir uma frase sobre cada animal, por exemplo, *Um cachorro seria um ótimo animal de estimação.*
- Acrescente uma linha no final de seu programa informando o que esses animais têm em comum. Você poderia exibir uma frase como *Qualquer um desses animais seria um ótimo animal de estimação!*

Criando listas numéricas

Há muitos motivos para armazenar um conjunto de números. Por exemplo, você precisará manter um controle das posições de cada personagem em um jogo, e talvez queira manter um registro das pontuações mais altas de um jogador também. Em visualizações de dados,

quase sempre você trabalhará com conjuntos de números, como temperaturas, distâncias, tamanhos de população ou valores de latitudes e longitudes, entre outros tipos de conjuntos numéricos.

As listas são ideais para armazenar conjuntos de números, e Python oferece várias ferramentas para ajudar você a trabalhar com listas de números de forma eficiente. Depois que souber usar efetivamente essas ferramentas, seu código funcionará bem, mesmo quando suas listas tiverem milhões de itens.

Usando a função range()

A função `range()` de Python facilita gerar uma série de números. Por exemplo, podemos usar a função `range()` para exibir uma sequência de números, assim:

numbers.py

```
for value in range(1,5):
    print(value)
```

Embora esse código dê a impressão de que deveria exibir os números de 1 a 5, ele não exibe o número 5:

```
1
2
3
4
```

Nesse exemplo, `range()` exibe apenas os números de 1 a 4. Esse é outro resultado do comportamento deslocado de um que veremos com frequência nas linguagens de programação. A função `range()` faz Python começar a contar no primeiro valor que você lhe fornecer e parar quando atingir o segundo valor especificado. Como ele para nesse segundo valor, a saída não conterá o valor final, que seria 5, nesse caso.

Para exibir os números de 1 a 5, você deve usar `range(1,6)`:

```
for value in range(1,6):
    print(value)
```

Dessa vez, a saída começa em 1 e termina em 5:

```
1
2
3
4
5
```

Se sua saída for diferente do esperado ao usar `range()`, experimente ajustar seu valor final em 1.

Usando range() para criar uma lista de números

Se quiser criar uma lista de números, você pode converter os resultados de `range()` diretamente em uma lista usando a função `list()`. Quando colocamos `list()` em torno de uma chamada à função `range()`, a saída será uma lista de números.

No exemplo da seção anterior, simplesmente exibimos uma série de números. Podemos usar `list()` para converter esse mesmo conjunto de números em uma lista:

```
numbers = list(range(1,6))
print(numbers)
```

E o resultado será este:

```
[1, 2, 3, 4, 5]
```

Também podemos usar a função `range()` para dizer a Python que ignore alguns números em um dado intervalo. Por exemplo, eis o modo de listar os números pares entre 1 e 10:

even_numbers.py

```
even_numbers = list(range(2,11,2))
print(even_numbers)
```

Nesse exemplo, a função `range()` começa com o valor 2 e então soma 2 a esse valor. O valor 2 é somado repetidamente até o valor final, que é 11, ser alcançado ou ultrapassado, e o resultado a seguir é gerado:

```
[2, 4, 6, 8, 10]
```

Podemos criar praticamente qualquer conjunto de números que

quisermos com a função `range()`. Por exemplo, considere como criaríamos uma lista dos dez primeiros quadrados perfeitos (isto é, o quadrado de cada inteiro de 1 a 10). Em Python, dois asteriscos (`**`) representam exponenciais. Eis o modo como podemos colocar os dez primeiros quadrados perfeitos em uma lista:

squares.py

```
u squares = []
v for value in range(1,11):
w     square = value**2
x     squares.append(square)

y print(squares)
```

Começamos com uma lista vazia chamada `squares` em u. Em v, dizemos a Python para percorrer cada valor de 1 a 10 usando a função `range()`. No laço, o valor atual é elevado ao quadrado e armazenado na variável `square` em w. Em x, cada novo valor de `square` é concatenado à lista `squares`. Por fim, quando o laço acaba de executar, a lista de quadrados é exibida em y:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Para escrever esse código de modo mais conciso, omita a variável temporária `square` e concatene cada novo valor diretamente na lista:

```
  squares = []
  for value in range(1,11):
u     squares.append(value**2)

  print(squares)
```

O código em u faz a mesma tarefa executada pelas linhas w e x em *squares.py*. Cada valor do laço é elevado ao quadrado e, então, é imediatamente concatenado à lista de quadrados.

Você pode usar qualquer uma dessas duas abordagens quando criar listas mais complexas. Às vezes, usar uma variável temporária deixa o código mais legível; em outras ocasiões, deixa o código desnecessariamente longo. Concentre-se primeiro em escrever um código que você entenda claramente e faça o que você quer que ele faça. Em seguida, procure

abordagens mais eficientes à medida que revisar seu código.

Estatísticas simples com uma lista de números

Algumas funções Python são específicas para listas de números. Por exemplo, podemos encontrar facilmente o valor mínimo, o valor máximo e a soma de uma lista de números:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTA Os exemplos desta seção utilizam listas pequenas de números para que caibam facilmente na página. Esses exemplos também funcionarão bem se sua lista contiver um milhão de números ou mais.

List comprehensions

A abordagem descrita antes para gerar a lista `squares` usou três ou quatro linhas de código. Uma *list comprehension* (abrangência de lista) permite gerar essa mesma lista com apenas uma linha de código. Uma list comprehension combina o laço `for` e a criação de novos elementos em uma linha, e concatena cada novo elemento automaticamente. As list comprehensions nem sempre são apresentadas aos iniciantes, mas eu as incluí aqui porque é bem provável que você as veja assim que começar a analisar códigos de outras pessoas.

O exemplo a seguir cria a mesma lista de quadrados perfeitos que vimos antes, porém utiliza uma list comprehension:

squares.py

```
squares = [value**2 for value in range(1,11)]
print(squares)
```

Para usar essa sintaxe, comece com um nome descritivo para a lista, por exemplo, `squares`. Em seguida, insira um colchete de abertura e defina a

expressão para os valores que você quer armazenar na nova lista. Nesse exemplo, a expressão é `value**2`, que eleva o valor ao quadrado. Então escreva um laço `for` para gerar os números que você quer fornecer à expressão e insira um colchete de fechamento. O laço `for` nesse exemplo é `for value in range(1,11)`, que fornece os valores de 1 a 10 à expressão `value**2`. Observe que não usamos dois-pontos no final da instrução `for`.

O resultado é a mesma lista de valores ao quadrado que vimos antes:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Escrever suas próprias list comprehensions exige um pouco de prática, mas você verá que vale a pena conhecê-las depois que se sentir à vontade para criar listas comuns. Quando escrever três ou quatro linhas de código para gerar listas e isso começar a parecer repetitivo, considere escrever suas próprias list comprehensions.

FAÇA VOCÊ MESMO

4.3 – Contando até vinte: Use um laço `for` para exibir os números de 1 a 20, incluindo-os.

4.4 – Um milhão: Crie uma lista de números de um a um milhão e, então, use um laço `for` para exibir os números. (Se a saída estiver demorando demais, interrompa pressionando CTRL-C ou feche a janela de saída.)

4.5 – Somando um milhão: Crie uma lista de números de um a um milhão e, em seguida, use `min()` e `max()` para garantir que sua lista realmente começa em um e termina em um milhão. Além disso, utilize a função `sum()` para ver a rapidez com que Python é capaz de somar um milhão de números.

4.6 – Números ímpares: Use o terceiro argumento da função `range()` para criar uma lista de números ímpares de 1 a 20. Utilize um laço `for` para exibir todos os números.

4.7– Três: Crie uma lista de múltiplos de 3, de 3 a 30. Use um laço `for` para exibir os números de sua lista.

4.8 – Cubos: Um número elevado à terceira potência é chamado de *cubo*. Por exemplo, o cubo de 2 é escrito como `2**3` em Python. Crie uma lista dos dez primeiros cubos (isto é, o cubo de cada inteiro de 1 a 10), e utilize um laço `for` para exibir o valor de cada cubo.

4.9 – Comprehension de cubos: Use uma list comprehension para gerar uma lista dos dez primeiros cubos.

Trabalhando com parte de uma lista

No Capítulo 3 aprendemos a acessar elementos únicos de uma lista e, neste capítulo, aprendemos a trabalhar com todos os elementos de uma lista. Também podemos trabalhar com um grupo específico de itens de uma lista, que Python chama de *fatia*.

Fatiando uma lista

Para criar uma fatia, especifique o índice do primeiro e do último elemento com os quais você quer trabalhar. Como ocorre na função `range()`, Python para em um item antes do segundo índice que você especificar. Para exibir os três primeiros elementos de uma lista, solicite os índices de `0` a `3`; os elementos `0`, `1` e `2` serão devolvidos.

O exemplo a seguir envolve uma lista de jogadores de um time:

players.py

```
  players = ['charles', 'martina', 'michael', 'florence', 'eli']
u print(players[0:3])
```

O código em u exibe uma fatia dessa lista, que inclui apenas os três primeiros jogadores. A saída mantém a estrutura de lista e inclui os três primeiros jogadores:

```
['charles', 'martina', 'michael']
```

Você pode gerar qualquer subconjunto de uma lista. Por exemplo, se quiser o segundo, o terceiro e o quarto itens de uma lista, comece a fatia no índice `1` e termine no índice `4`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

Dessa vez, a fatia começa com `'martina'` e termina com `'florence'`:

```
['martina', 'michael', 'florence']
```

Se o primeiro índice de uma fatia for omitido, Python começará sua fatia automaticamente no início da lista:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Sem um índice de início, Python usa o início da lista:

```
['charles', 'martina', 'michael', 'florence']
```

Uma sintaxe semelhante funcionará se você quiser uma fatia que inclua o final de uma lista. Por exemplo, se quiser todos os itens do terceiro até o último item, podemos começar com o índice `2` e omitir o segundo índice:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python devolve todos os itens, do terceiro item até o final da lista:

```
['michael', 'florence', 'eli']
```

Essa sintaxe permite apresentar todos os elementos a partir de qualquer ponto de sua lista até o final, independentemente do tamanho da lista. Lembre-se de que um índice negativo devolve um elemento a uma determinada distância do final de uma lista; assim, podemos exibir qualquer fatia a partir do final de uma lista. Por exemplo, se quisermos apresentar os três últimos jogadores da lista, podemos usar a fatia `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

Esse código exibe os nomes dos três últimos jogadores e continuaria a funcionar à medida que a lista de jogadores mudar de tamanho.

Percorrendo uma fatia com um laço

Você pode usar uma fatia em um laço `for` se quiser percorrer um subconjunto de elementos de uma lista. No próximo exemplo, percorreremos os três primeiros jogadores e exibiremos seus nomes como parte de uma lista simples:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
```

```
u for player in players[:3]:
    print(player.title())
```

Em vez de percorrer a lista inteira de jogadores em u, Python percorre somente os três primeiros nomes:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

As fatias são muito úteis em várias situações. Por exemplo, quando criar um jogo, você poderia adicionar a pontuação final de um jogador em uma lista sempre que esse jogador acabar de jogar. Seria possível então obter as três pontuações mais altas de um jogador ordenando a lista em ordem decrescente e obtendo uma lista que inclua apenas as três primeiras pontuações. Ao trabalhar com dados, você pode usar fatias para processar seus dados em porções de tamanho específico. Quando criar uma aplicação web, fatias poderiam ser usadas para exibir informações em uma série de páginas, com uma quantidade apropriada de informações em cada página.

Copiando uma lista

Com frequência, você vai querer começar com uma lista existente e criar uma lista totalmente nova com base na primeira. Vamos explorar o modo de copiar uma lista e analisar uma situação em que copiar uma lista é útil.

Para copiar uma lista, podemos criar uma fatia que inclua a lista original inteira omitindo o primeiro e o segundo índices (`[:]`). Isso diz a Python para criar uma lista que começa no primeiro item e termina no último, gerando uma cópia da lista toda.

Por exemplo, suponha que temos uma lista de nossos alimentos prediletos e queremos criar uma lista separada de comidas que um amigo gosta. Esse amigo gosta de tudo que está em nossa lista até agora, portanto podemos criar sua lista copiando a nossa:

foods.py

```
u my_foods = ['pizza', 'falafel', 'carrot cake']
v friend_foods = my_foods[:]

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Em u criamos uma lista de alimentos de que gostamos e a chamamos de `my_foods`. Em v criamos uma nova lista chamada `friend_foods`. Fizemos uma cópia de `my_foods` solicitando uma fatia de `my_foods` sem especificar qualquer índice e armazenamos a cópia em `friend_foods`. Quando exibimos cada lista, vemos que as duas contêm os mesmos alimentos:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

Para provar que realmente temos duas listas separadas, acrescentaremos um alimento em cada lista e mostraremos que cada lista mantém um registro apropriado das comidas favoritas de cada pessoa:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']
u friend_foods = my_foods[:]

v my_foods.append('cannoli')
w friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Em u, copiamos os itens originais em `my_foods` para a nova lista `friend_foods`, como fizemos no exemplo anterior. Em seguida, adicionamos um novo alimento em cada lista: em v, acrescentamos `'cannoli'` a `my_foods` e em w, adicionamos `'ice cream'` em `friend_foods`. Em seguida, exibimos as

duas listas para ver se cada um desses alimentos está na lista apropriada.

```
  My favorite foods are:
x ['pizza', 'falafel', 'carrot cake', 'cannoli']

  My friend's favorite foods are:
y ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

A saída em x mostra que `'cannoli'` agora aparece em nossa lista de alimentos prediletos, mas `'ice cream'` não. Em y podemos ver que `'ice cream'` agora aparece na lista de nosso amigo, mas `'cannoli'` não. Se tivéssemos simplesmente definido `friend_foods` como igual a `my_foods`, não teríamos gerado duas listas separadas. Por exemplo, eis o que acontece quando tentamos copiar uma lista sem usar uma fatia:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']

  # Isto não funciona:
u friend_foods = my_foods

  my_foods.append('cannoli')
  friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Em vez de armazenar uma cópia de `my_foods` em `friend_foods` em u, definimos que `friend_foods` é igual a `my_foods`. Essa sintaxe, na verdade, diz a Python para conectar a nova variável `friend_foods` à lista que já está em `my_foods`, de modo que, agora, as duas variáveis apontam para a mesma lista. Como resultado, quando adicionamos `'cannoli'` em `my_foods`, essa informação aparecerá em `friend_foods`. De modo semelhante, `'ice cream'` aparecerá nas duas listas, apesar de parecer que foi adicionada somente em `friend_foods`.

A saída mostra que as duas listas agora são iguais, mas não é isso que queríamos fazer:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

NOTA Não se preocupe com os detalhes desse exemplo por enquanto. Basicamente, se você estiver tentando trabalhar com uma cópia de uma lista e vir um comportamento inesperado, certifique-se de que está copiando a lista usando uma fatia, como fizemos no primeiro exemplo.

FAÇA VOCÊ MESMO

4.10 – Fatias: Usando um dos programas que você escreveu neste capítulo, acrescente várias linhas no final do programa que façam o seguinte:

- Exiba a mensagem *Os três primeiros itens da lista são:* Em seguida, use uma fatia para exibir os três primeiros itens da lista desse programa.
- Exiba a mensagem *Três itens do meio da lista são:*. Use uma fatia para exibir três itens do meio da lista.
- Exiba a mensagem *Os três últimos itens da lista são:*. Use uma fatia para exibir os três últimos itens da lista.

4.11 – Minhas pizzas, suas pizzas: Comece com seu programa do Exercício 4.1 (página 97). Faça uma cópia da lista de pizzas e chame-a de `friend_pizzas`. Então faça o seguinte:

- Adicione uma nova pizza à lista original.
- Adicione uma pizza diferente à lista `friend_pizzas`.
- Prove que você tem duas listas diferentes. Exiba a mensagem *Minhas pizzas favoritas são:; em seguida, utilize um laço* `for` *para exibir a primeira lista*. Exiba a mensagem *As pizzas favoritas de meu amigo são:*; em seguida, utilize um laço `for` para exibir a segunda lista. Certifique-se de que cada pizza nova esteja armazenada na lista apropriada.

4.12 – Mais laços: Todas as versões de *foods.py* nesta seção evitaram usar laços `for` para fazer exibições a fim de economizar espaço. Escolha uma versão de *foods.py* e escreva dois laços `for` para exibir cada lista de comidas.

Tuplas

As listas funcionam bem para armazenar conjuntos de itens que podem mudar durante a vida de um programa. A capacidade de modificar listas é

particularmente importante quando trabalhamos com uma lista de usuários em um site ou com uma lista de personagens em um jogo. No entanto, às vezes, você vai querer criar uma lista de itens que não poderá mudar. As tuplas permitir fazer exatamente isso. Python refere-se a valores que não podem mudar como *imutáveis*, e uma lista imutável é chamada de *tupla*.

Definindo uma tupla

Uma tupla se parece exatamente com uma lista, exceto por usar parênteses no lugar de colchetes. Depois de definir uma tupla, podemos acessar elementos individuais usando o índice de cada item, como faríamos com uma lista.

Por exemplo, se tivermos um retângulo que sempre deva ter determinado tamanho, podemos garantir que seu tamanho não mudará colocando as dimensões em uma tupla:

dimensions.py

```
u dimensions = (200, 50)
v print(dimensions[0])
  print(dimensions[1])
```

Definimos a tupla `dimensions` em u, usando parênteses no lugar de colchetes. Em v exibimos cada elemento da tupla individualmente com a mesma sintaxe que usamos para acessar elementos de uma lista:

```
200
50
```

Vamos ver o que acontece se tentarmos alterar um dos itens da tupla `dimensions`:

```
  dimensions = (200, 50)
u dimensions[0] = 250
```

O código em u tenta mudar o valor da primeira dimensão, mas Python devolve um erro de tipo. Basicamente, pelo fato de estarmos tentando alterar uma tupla, o que não pode ser feito para esse tipo de objeto, Python nos informa que não podemos atribuir um novo valor a um item em uma

tupla:

```
Traceback (most recent call last):
  File "dimensions.py", line 3, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

Isso é um ponto positivo, pois queremos que Python gere um erro se uma linha de código tentar alterar as dimensões do retângulo.

Percorrendo todos os valores de uma tupla com um laço

Podemos percorrer todos os valores de uma tupla usando um laço `for`, assim como fizemos com uma lista:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python devolve todos os elementos da tupla, como faria com uma lista:

```
200
50
```

Sobrescrevendo uma tupla

Embora não seja possível modificar uma tupla, podemos atribuir um novo valor a uma variável que armazena uma tupla. Portanto, se quiséssemos alterar nossas dimensões, poderíamos redefinir a tupla toda:

```
u dimensions = (200, 50)
  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

v dimensions = (400, 100)
w print("\nModified dimensions:")
  for dimension in dimensions:
      print(dimension)
```

O bloco em u define a tupla original e exibe as dimensões iniciais. Em v armazenamos uma nova tupla na variável `dimensions`. Em seguida, exibimos

as novas dimensões em `w`. Python não gera nenhum erro dessa vez, pois sobrescrever uma variável é uma operação válida:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

Se comparada com listas, as tuplas são estruturas de dados simples. Use-as quando quiser armazenar um conjunto de valores que não deva ser alterado durante a vida de um programa.

FAÇA VOCÊ MESMO

4.13 – Buffet: Um restaurante do tipo buffet oferece apenas cinco tipos básicos de comida. Pense em cinco pratos simples e armazene-os em uma tupla.

- Use um laço `for` para exibir cada prato oferecido pelo restaurante.
- Tente modificar um dos itens e cerifique-se de que Python rejeita a mudança.
- O restaurante muda seu cardápio, substituindo dois dos itens com pratos diferentes. Acrescente um bloco de código que reescreva a tupla e, em seguida, use um laço `for` para exibir cada um dos itens do cardápio revisado.

Estilizando seu código

Agora que você está escrevendo programas mais longos, vale a pena conhecer algumas ideias sobre como estilizar seu código. Reserve tempo para deixar seu código o mais legível possível. Escrever códigos fáceis de ler ajuda a manter o controle sobre o que seus programas fazem e também contribui para que outras pessoas entendam seu código.

Programadores Python chegaram a um acordo sobre várias convenções de estilo a fim de garantir que o código de todos, de modo geral, esteja estruturado da mesma maneira. Depois que aprender a escrever código Python limpo, você deverá ser capaz de entender a estrutura geral do código Python de qualquer pessoa, desde que elas sigam as mesmas

diretrizes. Se você espera tornar-se um programador profissional em algum momento, comece a seguir essas diretrizes o mais rápido possível, a fim de desenvolver bons hábitos.

Guia de estilo

Quando alguém quer fazer uma alteração na linguagem Python, essa pessoa escreve uma *PEP* (Python Enhancement Proposal, ou Proposta de Melhoria de Python). Uma das PEPs mais antigas é a *PEP 8*, que instrui programadores Python a estilizar seu código. A PEP é bem longa, mas boa parte dela está relacionada a estruturas de código mais complexas do que vimos até agora.

O guia de estilo de Python foi escrito considerando que o código é lido com mais frequência que é escrito. Você escreverá seu código uma vez e então começará a lê-lo quando começar a depuração. Quando acrescentar recursos a um programa, você gastará mais tempo lendo o seu código. Ao compartilhar seu código com outros programadores, eles também o lerão.

Dada a opção entre escrever um código que seja mais fácil de escrever ou um código que seja mais fácil de ler, os programadores Python quase sempre incentivarão você a escrever um código que seja mais fácil de ler. As diretrizes a seguir ajudarão a escrever um código claro desde o início.

Indentação

A PEP 8 recomenda usar quatro espaços por nível de indentação. Usar quatro espaços melhora a legibilidade, ao mesmo tempo que deixa espaço para vários níveis de indentação em cada linha.

Em um documento de processador de texto, com frequência, as pessoas usam tabulações no lugar de espaços para indentar. Isso funciona bem para documentos de processadores de texto, mas o interpretador Python fica confuso quando tabulações são misturadas a espaços. Todo editor de texto oferece uma configuração que permite usar a tecla TAB, mas converte cada tabulação em um número definido de espaços. Definitivamente, você deve usar a tecla TAB, mas certifique-se também de que seu editor esteja configurado para inserir espaços no lugar de tabulações em seu

documento.

Misturar tabulações com espaços em seu arquivo pode causar problemas que são difíceis de diagnosticar. Se você achar que tem uma mistura de tabulações e espaços, é possível converter todas as tabulações de um arquivo em espaços na maioria dos editores.

Tamanho da linha

Muitos programadores Python recomendavam que cada linha deveria ter menos de 80 caracteres. Historicamente, essa diretriz se desenvolveu porque a maioria dos computadores conseguia inserir apenas 79 caracteres em uma única linha em uma janela de terminal. Atualmente, as pessoas conseguem colocar linhas mais longas em suas telas, mas há outros motivos para se ater ao tamanho de linha-padrão de 79 caracteres. Programadores profissionais muitas vezes têm vários arquivos abertos na mesma tela, e usar o tamanho-padrão de linha lhes permite ver linhas inteiras em dois ou três arquivos abertos lado a lado na tela. A PEP 8 também recomenda que você limite todos os seus comentários em 72 caracteres por linha, pois algumas das ferramentas que geram documentação automática para projetos maiores acrescentam caracteres de formatação no início de cada linha comentada.

As diretrizes da PEP 8 para tamanho de linha não estão gravadas a ferro e fogo, e algumas equipes preferem um limite de 99 caracteres. Não se preocupe demais com o tamanho da linha em seu código enquanto estiver aprendendo, mas saiba que as pessoas que trabalham de modo colaborativo quase sempre seguem as orientações da PEP 8. A maioria dos editores permite configurar uma indicação visual – geralmente é uma linha vertical na tela – que mostra em que ponto estão esses limites.

NOTA O Apêndice B mostra como configurar seu editor de texto de modo que ele insira quatro espaços sempre que você pressionar a tecla TAB e mostre uma linha vertical para ajudá-lo a seguir o limite de 79 caracteres.

Linhas em branco

Para agrupar partes de seu programa visualmente, utilize linhas em branco.

Você deve usar linhas em branco para organizar seus arquivos, mas não as use de modo excessivo. Ao seguir os exemplos fornecidos neste livro, você deverá atingir o equilíbrio adequado. Por exemplo, se você tiver cinco linhas de código que criem uma lista e, então, outras três linhas que façam algo com essa lista, será apropriado colocar uma linha em branco entre as duas seções. Entretanto, você não deve colocar três ou quatro linhas em branco entre as duas seções.

As linhas em branco não afetarão o modo como seu código é executado, mas influenciarão em sua legibilidade. O interpretador Python utiliza indentação horizontal para interpretar o significado de seu código, mas despreza o espaçamento vertical.

Outras diretrizes de estilo

A PEP 8 tem muitas recomendações adicionais para estilo, mas a maioria das diretrizes refere-se a programas mais complexos que aqueles que você está escrevendo no momento. À medida que conhecer estruturas Python mais complexas, compartilharei as partes relevantes das diretrizes da PEP 8.

FAÇA VOCÊ MESMO

4.14 – PEP 8: Observe o guia de estilo da PEP 8 original em *https://python.org/dev/peps/pep-0008/*. Você não usará boa parte dele agora, mas pode ser interessante dar uma olhada.

4.15 – Revisão de código: Escolha três programas que você escreveu neste capítulo e modifique-os para que estejam de acordo com a PEP 8:

- Use quatro espaços para cada nível de indentação. Configure seu editor de texto para inserir quatro espaços sempre que a tecla TAB for usada, caso ainda não tenha feito isso (consulte o Apêndice B para ver instruções sobre como fazê-lo).
- Use menos de 80 caracteres em cada linha e configure seu editor para que mostre uma linha vertical na posição do caractere de número 80.
- Não use linhas em branco em demasia em seus arquivos de programa.

Resumo

Neste capítulo aprendemos a trabalhar de modo eficiente com os elementos de uma lista. Vimos como percorrer uma lista usando um laço `for`, como Python usa indentação para estruturar um programa e como evitar alguns erros comuns de indentação. Aprendemos a criar listas numéricas simples, além de conhecermos algumas operações que podem ser realizadas em listas numéricas. Aprendemos a fatiar uma lista para trabalhar com um subconjunto de itens e a copiar listas de modo adequado usando uma fatia. Também conhecemos as tuplas, que oferecem um nível de proteção a um conjunto de valores que não deve ser alterado, e aprendemos a estilizar seu código cada vez mais complexo para facilitar sua leitura.

No Capítulo 5 veremos como responder de forma apropriada a diferentes condições usando instruções `if`. Aprenderemos a combinar conjuntos relativamente complexos de testes condicionais para responder de forma apropriada ao tipo de situação desejada ou oferecer informações que você esteja procurando. Também aprenderemos a usar instruções `if` enquanto percorremos uma lista para realizar ações específicas com elementos selecionados de uma lista.

5

INSTRUÇÕES IF

Com frequência, a programação envolve analisar um conjunto de condições e decidir qual ação deve ser executada de acordo com essas condições. A instrução `if` de Python permite analisar o estado atual de um programa e responder de forma apropriada a esse estado.

Neste capítulo aprenderemos a escrever testes condicionais, que permitem verificar qualquer condição que seja de seu interesse. Veremos como escrever instruções `if` simples e criar uma série de instruções `if` mais complexa para identificar se as condições exatas que você quer estão presentes. Então você aplicará esse conceito a listas, de modo que será possível escrever um laço `for` que trate a maioria dos itens de uma lista de uma maneira, mas determinados itens com valores específicos de modo diferente.

Um exemplo simples

O pequeno exemplo a seguir mostra como testes `if` permitem responder a situações especiais de forma correta. Suponha que você tenha uma lista de carros e queira exibir o nome de cada carro. Carros têm nomes próprios, portanto a maioria deles deve ser exibido com a primeira letra maiúscula. No entanto, o valor `'bmw'` deve ser apresentado somente com letras maiúsculas. O código a seguir percorre uma lista de nomes de carros em um laço e procura o valor `'bmw'`. Sempre que o valor for `'bmw'`, ele será exibido com letras maiúsculas, e não apenas com a inicial maiúscula:

cars.py

```
  cars = ['audi', 'bmw', 'subaru', 'toyota']

  for car in cars:
u     if car == 'bmw':
          print(car.upper())
      else:
```

```
        print(car.title())
```

O laço nesse exemplo inicialmente verifica se o valor atual de `car` é `'bmw'` u. Em caso afirmativo, o valor é exibido com letras maiúsculas. Se o valor de `car` for diferente de `'bmw'`, será exibido com a primeira letra maiúscula:

```
Audi
BMW
Subaru
Toyota
```

Esse exemplo combina vários conceitos que veremos neste capítulo. Vamos começar observando os tipos de teste que podemos usar para analisar as condições em seu programa.

Testes condicionais

No coração de cada instrução `if` está uma expressão que pode ser avaliada como `True` ou `False`, chamada de *teste condicional*. Python usa os valores `True` e `False` para decidir se o código em uma instrução `if` deve ser executado. Se um teste condicional for avaliado como `True`, Python executará o código após a instrução `if`. Se o teste for avaliado como `False`, o interpretador ignorará o código depois da instrução `if`.

Verificando a igualdade

A maioria dos testes condicionais compara o valor atual de uma variável com um valor específico de interesse. O teste condicional mais simples verifica se o valor de uma variável é igual ao valor de interesse:

```
u >>> car = 'bmw'
v >>> car == 'bmw'
  True
```

A linha em u define o valor de `car` como `'bmw'` usando um único sinal de igualdade, como já vimos muitas vezes. A linha em v verifica se o valor de `car` é `'bmw'` usando um sinal de igualdade duplo (==). Esse *operador de igualdade* devolve `True` se os valores dos lados esquerdo e direito do

operador forem iguais, e `False` se forem diferentes. Os valores nesse exemplo são iguais, portanto Python devolve `True`.

Quando o valor de `car` for diferente de `'bmw'`, esse teste devolverá `False`:

```
u >>> car = 'audi'
v >>> car == 'bmw'
  False
```

Um único sinal de igual, na verdade, é uma instrução; você poderia ler o código em **u** como "defina o valor de `car` como `'audi'`". Por outro lado, um sinal de igualdade duplo, como o que está em **v**, faz uma pergunta: "O valor de `car` é igual a `'bmw'`?". A maioria das linguagens de programação utiliza sinais de igualdade dessa maneira.

Ignorando as diferenças entre letras maiúsculas e minúsculas ao verificar a igualdade

Testes de igualdade diferenciam letras maiúsculas de minúsculas em Python. Por exemplo, dois valores com diferenças apenas quanto a letras maiúsculas ou minúsculas não são considerados iguais:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

Se a diferença entre letras maiúsculas e minúsculas for importante, esse comportamento será vantajoso. Porém, se essa diferença não importar e você simplesmente quiser testar o valor de uma variável, poderá converter esse valor para letras minúsculas antes de fazer a comparação:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Esse teste deve devolver `True`, independentemente do modo como o valor `'Audi'` estiver formatado, pois o teste agora ignora as diferenças entre letras maiúsculas e minúsculas. A função `lower()` não altera o valor originalmente armazenado em `car`, portanto você pode fazer esse tipo de comparação sem afetá-lo:

```
u >>> car = 'Audi'
v >>> car.lower() == 'audi'
  True
w >>> car
  'Audi'
```

Em u armazenamos a string `'Audi'` com a primeira letra maiúscula na variável `car`. Em v convertemos o valor de `car` para letras minúsculas e comparamos esse valor com a string `'audi'`. As duas strings são iguais, portanto Python devolve `True`. Em w podemos ver que o valor armazenado em `car` não foi afetado pelo teste condicional.

Os sites impõem determinadas regras para os dados fornecidos pelos usuários de modo semelhante a esse. Por exemplo, um site pode usar um teste condicional desse tipo para garantir que todos os usuários tenham um nome realmente único, e não apenas uma variação quanto ao uso de letras maiúsculas em relação ao nome de usuário de outra pessoa. Quando alguém submeter um novo nome de usuário, esse nome será convertido para letras minúsculas e comparado às versões com letras minúsculas de todos os nomes de usuário existentes. Durante essa verificação, um nome de usuário como `'John'` será rejeitado se houver qualquer variação de `'john'` já em uso.

Verificando a diferença

Se quiser determinar se dois valores não são iguais, você poderá combinar um ponto de exclamação e um sinal de igualdade (`!=`). O ponto de exclamação representa *não*, como ocorre em muitas linguagens de programação.

Vamos usar outra instrução `if` para ver como o operador "diferente de" é usado. Armazenaremos o ingrediente pedido em uma pizza em uma variável e então exibiremos uma mensagem se a pessoa não pediu anchovas:

toppings.py

```
  requested_topping = 'mushrooms'

u if requested_topping != 'anchovies':
```

```
    print("Hold the anchovies!")
```

A linha em u compara o valor de `requested_topping` com o valor `'anchovies'`. Se esses dois valores não forem iguais, Python devolverá `True` e executará o código após a instrução `if`. Se esses dois valores forem iguais, Python devolverá `False` e não executará o código após essa instrução.

Como o valor de `requested_topping` não é `'anchovies'`, a instrução `print` é executada:

```
Hold the anchovies!
```

A maior parte das expressões condicionais que você escrever testará a igualdade, porém, às vezes, você achará mais eficiente testar a não igualdade.

Comparações numéricas

Testar valores numéricos é bem simples. Por exemplo, o código a seguir verifica se uma pessoa tem 18 anos:

```
>>> age = 18
>>> age == 18
True
```

Também podemos testar se dois números não são iguais. Por exemplo, o código a seguir exibe uma mensagem se a resposta dada não estiver correta:

magic_number.py

```
  answer = 17

u if answer != 42:
      print("That is not the correct answer. Please try again!")
```

O teste condicional em u passa porque o valor de `answer` (`17`) não é igual a `42`. Como o teste passa, o bloco de código indentado é executado:

```
That is not the correct answer. Please try again!
```

Você pode incluir também várias comparações matemáticas em suas instruções condicionais, por exemplo, menor que, menor ou igual a, maior

que e maior ou igual a:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Toda comparação matemática pode ser usada como parte de uma instrução `if`, o que pode ajudar você a determinar as condições exatas de interesse.

Testando várias condições

Pode ser que você queira testar várias condições ao mesmo tempo. Por exemplo, ocasionalmente, você pode precisar de duas condições `True` para executar uma ação. Em outros casos, poderia ficar satisfeito com apenas uma condição `True`. As palavras reservadas `and` e `or` podem ajudar nessas situações.

Usando and para testar várias condições

Para verificar se duas condições são `True` simultaneamente, utilize a palavra reservada `and` para combinar os dois testes condicionais; se cada um dos testes passar, a expressão como um todo será avaliada como `True`. Se um dos testes falhar, ou ambos, a expressão será avaliada como `False`.

Por exemplo, podemos verificar se duas pessoas têm mais de 21 anos usando o teste a seguir:

```
u >>> age_0 = 22
  >>> age_1 = 18
v >>> age_0 >= 21 and age_1 >= 21
  False
w >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

Em u definimos duas idades, age_0 e age_1. Em v verificamos se as duas idades são maiores ou iguais a 21. O teste à esquerda passa, porém, o teste à direita falha, portanto a expressão condicional como um todo é avaliada como False. Em w mudamos age_1 para 22. O valor de age_1 agora é maior que 21, portanto os dois testes individuais passam, fazendo com que a expressão condicional como um todo seja avaliada como True.

Para melhorar a legibilidade, podemos usar parênteses em torno dos testes individuais, mas eles não são obrigatórios. Se usar parênteses, seu teste terá o seguinte aspecto:

```
(age_0 >= 21) and (age_1 >= 21)
```

Usando or para testar várias condições

A palavra reservada or permite verificar várias condições também, mas o teste passa se um dos testes individuais passar, ou ambos. Uma expressão or falha somente quando os dois testes individuais falharem.

Vamos considerar duas idades novamente, porém, desta vez, queremos que apenas uma das pessoas tenha mais de 21 anos:

```
u >>> age_0 = 22
  >>> age_1 = 18
v >>> age_0 >= 21 or age_1 >= 21
  True
w >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

Começamos novamente com duas variáveis para idade em u. Como o teste para age_0 em v passa, a expressão com um todo é avaliada como True. Então diminuímos age_0 para 18. Em w os dois testes agora falham e a expressão como um todo é avaliada como False.

Verificando se um valor está em uma lista

Às vezes, é importante verificar se uma lista contém um determinado valor antes de executar uma ação. Por exemplo, talvez você queira verificar se um novo nome de usuário já existe em uma lista de nomes de usuários

atuais antes de concluir o registro de alguém em um site. Em um projeto de mapeamento, você pode querer verificar se uma localidade submetida já existe em uma lista de localidades conhecidas.

Para descobrir se um valor em particular já está em uma lista, utilize a palavra reservada `in`. Vamos observar um código que você poderia escrever para uma pizzaria. Criaremos uma lista de ingredientes que um cliente pediu em sua pizza e, então, verificaremos se determinados ingredientes estão na lista.

```
  >>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
u >>> 'mushrooms' in requested_toppings
  True
v >>> 'pepperoni' in requested_toppings
  False
```

Em u e em v, a palavra reservada `in` diz a Python para verificar a existência de `'mushrooms'` e de `'pepperoni'` na lista `requested_toppings`. Essa técnica é bem eficaz, pois podemos criar uma lista de valores essenciais e verificar facilmente se o valor que estamos testando é igual a um dos valores da lista.

Verificando se um valor não está em uma lista

Em outras ocasiões, é importante saber se um valor não está em uma lista. Podemos usar a palavra reservada `not` nesse caso. Por exemplo, considere uma lista de usuários que foi impedida de fazer comentários em um fórum. Podemos verificar se um usuário foi banido antes de permitir que essa pessoa submeta um comentário:

banned_users.py

```
  banned_users = ['andrew', 'carolina', 'david']
  user = 'marie'

u if user not in banned_users:
      print(user.title() + ", you can post a response if you wish.")
```

A linha em u é bem clara. Se o valor de `user` não estiver na lista `banned_users`, Python devolverá `True` e executará a linha indentada.

A usuária `'marie'` não está na lista `banned_users`, portanto ela vê uma mensagem convidando-a a postar uma resposta:

```
Marie, you can post a response if you wish.
```

Expressões booleanas

À medida que aprender mais sobre programação, você ouvirá o termo *expressão booleana* em algum momento. Uma expressão booleana é apenas outro nome para um teste condicional. Um *valor booleano* é `True` ou `False`, exatamente como o valor de uma expressão condicional após ter sido avaliada.

Valores booleanos muitas vezes são usados para manter o controle de determinadas condições, como o fato de um jogo estar executando ou se um usuário pode editar certos conteúdos em um site:

```
game_active = True
can_edit = False
```

Valores booleanos oferecem uma maneira eficiente para monitorar o estado ou uma condição em particular que seja importante para o seu programa.

FAÇA VOCÊ MESMO

5.1 – Testes condicionais: Escreva uma série de testes condicionais. Exiba uma frase que descreva o teste e o resultado previsto para cada um. Seu código deverá ser semelhante a:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Observe atentamente seus resultados e certifique-se de que compreende por que cada linha é avaliada como `True` ou `False`.
- Crie pelo menos dez testes. Tenha no mínimo cinco testes avaliados como `True` e outros cinco avaliados como `False`.

5.2 – Mais testes condicionais: Você não precisa limitar o número de testes que criar em dez. Se quiser testar mais comparações, escreva outros testes e acrescente-os em *conditional_tests.py*. Tenha pelo menos um resultado `True` e um `False` para cada um dos casos a seguir:

- testes de igualdade e de não igualdade com strings;
- testes usando a função `lower()`;
- testes numéricos que envolvam igualdade e não igualdade, maior e menor que, maior ou igual a e menor ou igual a;
- testes usando as palavras reservadas `and` e `or`;
- testes para verificar se um item está em uma lista;
- testes para verificar se um item não está em uma lista.

Instruções if

Quando compreender os testes condicionais, você poderá começar a escrever instruções `if`. Há vários tipos de instruções `if`, e a escolha de qual deles usar dependerá do número de condições que devem ser testadas. Vimos vários exemplos de instruções `if` na discussão sobre testes condicionais, mas agora vamos explorar mais o assunto.

Instruções if simples

O tipo mais simples de instrução `if` tem um teste e uma ação:

```
if teste_condicional:
    faça algo
```

Você pode colocar qualquer teste condicional na primeira linha, e praticamente qualquer ação no bloco indentado após o teste. Se o teste condicional for avaliado como `True`, Python executará o código após a instrução `if`. Se o teste for avaliado como `False`, Python ignorará o código depois da instrução `if`.

Suponha que temos uma variável que representa a idade de uma pessoa e queremos saber se essa pessoa tem idade suficiente para votar. O código a seguir testa se a pessoa pode votar:

voting.py

```
  age = 19
u if age >= 18:
v     print("You are old enough to vote!")
```

Em u Python verifica se o valor em `age` é maior ou igual a 18. É maior, portanto a instrução indentada `print` em v é executada:

```
You are old enough to vote!
```

A indentação tem a mesma função em instruções `if` que aquela desempenhada em laços `for`. Todas as linhas indentadas após uma instrução `if` serão executadas se o teste passar, e todo o bloco de linhas indentadas será ignorado se o teste não passar.

Podemos ter tantas linhas de código quantas quisermos no bloco após a instrução `if`. Vamos acrescentar outra linha de saída se a pessoa tiver idade suficiente para votar, perguntando se o indivíduo já se registrou para votar:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

O teste condicional passa e as duas instruções `print` estão indentadas, portanto ambas as linhas são exibidas:

```
You are old enough to vote!
Have you registered to vote yet?
```

Se o valor de `age` for menor que 18, esse programa não apresentará nenhuma saída.

Instruções if-else

Com frequência você vai querer executar uma ação quando um teste condicional passar, e uma ação diferente em todos os demais casos. A sintaxe `if-else` de Python torna isso possível. Um bloco `if-else` é semelhante a uma instrução `if` simples, porém a instrução `else` permite definir uma ação ou um conjunto de ações executado quando o teste condicional falhar.

Exibiremos a mesma mensagem mostrada antes se a pessoa tiver idade

suficiente para votar, porém, desta vez, acrescentaremos uma mensagem para qualquer um que não tenha idade suficiente para votar:

```
  age = 17
u if age >= 18:
      print("You are old enough to vote!")
      print("Have you registered to vote yet?")
v else:
      print("Sorry, you are too young to vote.")
      print("Please register to vote as soon as you turn 18!")
```

Se o teste condicional em u passar, o primeiro bloco de instruções `print` indentadas será executado. Se o teste for avaliado como `False`, o bloco `else` em v será executado. Como `age` é menor que 18 dessa vez, o teste condicional falha e o código do bloco `else` é executado:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

Esse código funciona porque há apenas duas possíveis situações para avaliar: uma pessoa tem idade suficiente para votar ou não tem. A estrutura `if-else` funciona bem em situações em que você quer que Python sempre execute uma de duas possíveis ações. Em uma cadeia `if-else` simples como essa, uma das duas ações sempre será executada.

Sintaxe if-elif-else

Muitas vezes, você precisará testar mais de duas situações possíveis; para avaliar isso, a sintaxe `if-elif-else` de Python poderá ser usada. Python executa apenas um bloco em uma cadeia `if-elif-else`. Cada teste condicional é executado em sequência, até que um deles passe. Quando um teste passar, o código após esse teste será executado e Python ignorará o restante dos testes.

Muitas situações do mundo real envolvem mais de duas condições possíveis. Por exemplo, considere um parque de diversões que cobre preços distintos para grupos etários diferentes:

- a entrada para qualquer pessoa com menos de 4 anos é gratuita;
- a entrada para qualquer pessoa com idade entre 4 e 18 anos custa 5

dólares;

• a entrada para qualquer pessoa com 18 anos ou mais custa 10 dólares.

Como podemos usar uma instrução `if` para determinar o preço da entrada de alguém? O código a seguir testa a faixa etária de uma pessoa e então exibe uma mensagem com o preço da entrada:

amusement_park.py

```
age = 12

u if age < 4:
     print("Your admission cost is $0.")
v elif age < 18:
     print("Your admission cost is $5.")
w else:
     print("Your admission cost is $10.")
```

O teste `if` em u verifica se uma pessoa tem menos de 4 anos. Se o teste passar, uma mensagem apropriada será exibida e Python ignorará o restante dos testes. A linha `elif` em v, na verdade, é outro teste `if`, executado somente se o teste anterior falhar. Nesse ponto da cadeia, sabemos que a pessoa tem pelo menos 4 anos de idade, pois o primeiro teste falhou. Se a pessoa tiver menos de 18 anos, uma mensagem apropriada será exibida, e Python ignorará o bloco `else`. Se tanto o teste em `if` quanto o teste em `elif` falharem, Python executará o código do bloco `else` em w.

Nesse exemplo, o teste em u é avaliado como `False`, portanto seu bloco de código não é executado. No entanto, o segundo teste é avaliado como `True` (12 é menor que 18), portanto seu código é executado. A saída é uma frase que informa o preço da entrada ao usuário:

```
Your admission cost is $5.
```

Qualquer idade acima de 17 anos faria os dois primeiros testes falharem. Nessas situações, o bloco `else` seria executado e o preço da entrada seria de 10 dólares.

Em vez de exibir o preço da entrada no bloco `if-elif-else`, seria mais conciso apenas definir o preço na cadeia `if-elif-else` e, então, ter uma

instrução `print` simples que execute depois que a cadeia for avaliada:

```
  age = 12

  if age < 4:
u     price = 0
  elif age < 18:
v     price = 5
  else:
w     price = 10

x print("Your admission cost is $" + str(price) + ".")
```

As linhas em u, v e w definem o valor de `price` conforme a idade da pessoa, como no exemplo anterior. Depois que o preço é definido pela cadeia `if-elif-else`, uma instrução `print` separada e não indentada x utiliza esse valor para exibir uma mensagem informando o preço de entrada da pessoa.

Esse código gera a mesma saída do exemplo anterior, mas o propósito da cadeia `if-elif-else` é mais restrito. Em vez de determinar um preço e exibir uma mensagem, ela simplesmente determina o preço da entrada. Além de ser mais eficiente, esse código revisado é mais fácil de modificar que a abordagem original. Para mudar o texto da mensagem de saída, seria necessário alterar apenas uma instrução `print`, e não três instruções `print` separadas.

Usando vários blocos elif

Podemos usar quantos blocos `elif` quisermos em nosso código. Por exemplo, se o parque de diversões implementasse um desconto para idosos, você poderia acrescentar mais um teste condicional no código a fim de determinar se uma pessoa está qualificada a receber esse desconto. Suponha que qualquer pessoa com 65 anos ou mais pague metade do preço normal da entrada, isto é, 5 dólares:

```
  age = 12

  if age < 4:
```

```
    price = 0
elif age < 18:
    price = 5
u elif age < 65:
    price = 10
v else:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

A maior parte desse código não foi alterada. O segundo bloco `elif` em u agora faz uma verificação para garantir que uma pessoa tenha menos de 65 anos antes de lhe cobrar o preço da entrada inteira, que é de 10 dólares. Observe que o valor atribuído no bloco `else` em v precisa ser alterado para 5 dólares, pois as únicas idades que chegam até esse bloco são de pessoas com 65 anos ou mais.

Omitindo o bloco else

Python não exige um bloco `else` no final de uma cadeia `if-elif`. Às vezes, um bloco `else` é útil; outras vezes, é mais claro usar uma instrução `elif` adicional que capture a condição específica de interesse:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
elif age < 65:
    price = 10
u elif age >= 65:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

O bloco `elif` extra em u atribui um preço de 5 dólares quando a pessoa tiver 65 anos ou mais, o que é um pouco mais claro que o bloco `else` geral. Com essa mudança, todo bloco de código deve passar por um teste específico para ser executado.

O bloco `else` é uma instrução que captura tudo. Ela corresponde a qualquer condição não atendida por um teste `if` ou `elif` específicos e isso, às vezes, pode incluir dados inválidos ou até mesmo maliciosos. Se você tiver uma condição final específica para testar, considere usar um último bloco `elif` e omitir o bloco `else`. Como resultado, você terá mais confiança de que seu código executará somente nas condições corretas.

Testando várias condições

A cadeia `if-elif-else` é eficaz, mas é apropriada somente quando você quiser que apenas um teste passe. Assim que encontrar um teste que passe, o interpretador Python ignorará o restante dos testes. Esse comportamento é vantajoso, pois é eficiente e permite testar uma condição específica.

Às vezes, porém, é importante verificar todas as condições de interesse. Nesse caso, você deve usar uma série de instruções `if` simples, sem blocos `elif` ou `else`. Essa técnica faz sentido quando mais de uma condição pode ser `True` e você quer atuar em todas as condições que sejam verdadeiras.

Vamos reconsiderar o exemplo da pizzaria. Se alguém pedir uma pizza com dois ingredientes, será necessário garantir que esses dois ingredientes sejam incluídos em sua pizza:

toppings.py

```
u requested_toppings = ['mushrooms', 'extra cheese']

v if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
w if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
x if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")

  print("\nFinished making your pizza!")
```

Começamos em u com uma lista contendo os ingredientes solicitados. A instrução `if` em v verifica se a pessoa pediu cogumelos (`'mushrooms'`) em sua pizza. Em caso afirmativo, uma mensagem será exibida para confirmar esse ingrediente. O teste para pepperoni em w corresponde a outra

instrução `if` simples, e não a uma instrução `elif` ou `else`, portanto esse teste é executado independentemente de o teste anterior ter passado ou não. O código em X verifica se queijo extra (`'extra cheese'`) foi pedido, não importando o resultado dos dois primeiros testes. Esses três testes independentes são realizados sempre que o programa é executado.

Como todas as condições nesse exemplo são avaliadas, tanto cogumelos quanto queijo extra são adicionados à pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Esse código não funcionaria de modo apropriado se tivéssemos usado um bloco `if-elif-else`, pois o código pararia de executar depois que apenas um teste tivesse passado. Esse código seria assim:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

O teste para `'mushrooms'` é o primeiro a passar, portanto cogumelos são adicionados à pizza. No entanto, os valores `'extra cheese'` e `'pepperoni'` não são verificados, pois Python não executa nenhum teste depois do primeiro que passar em uma cadeia `if-elif-else`. O primeiro ingrediente do cliente será adicionado, mas todos os demais não serão:

```
Adding mushrooms.

Finished making your pizza!
```

Em suma, se quiser que apenas um bloco de código seja executado, utilize uma cadeia `if-elif-else`. Se mais de um bloco de código deve

executar, utilize uma série de instruções `if` independentes.

FAÇA VOCÊ MESMO

5.3 – Cores de alienígenas #1: Suponha que um alienígena acabou de ser atingido em um jogo. Crie uma variável chamada **`alien_color`** e atribua-lhe um valor igual a **`'green'`**, **`'yellow'`** ou **`'red'`**.

- Escreva uma instrução **`if`** para testar se a cor do alienígena é verde. Se for, mostre uma mensagem informando que o jogador acabou de ganhar cinco pontos.
- Escreva uma versão desse programa em que o teste **`if`** passe e outro em que ele falhe. (A versão que falha não terá nenhuma saída.)

5.4 – Cores de alienígenas #2: Escolha uma cor para um alienígena, como foi feito no Exercício 5.3, e escreva uma cadeia **`if-else`**.

- Se a cor do alienígena for verde, mostre uma frase informando que o jogador acabou de ganhar cinco pontos por atingir o alienígena.
- Se a cor do alienígena não for verde, mostre uma frase informando que o jogador acabou de ganhar dez pontos.
- Escreva uma versão desse programa que execute o bloco **`if`** e outro que execute o bloco **`else`**.

5.5 – Cores de alienígenas #3: Transforme sua cadeia **`if-else`** do Exercício 5.4 em uma cadeia **`if-elif-else`**.

- Se o alienígena for verde, mostre uma mensagem informando que o jogador ganhou cinco pontos.
- Se o alienígena for amarelo, mostre uma mensagem informando que o jogador ganhou dez pontos.
- Se o alienígena for vermelho, mostre uma mensagem informando que o jogador ganhou quinze pontos.
- Escreva três versões desse programa, garantindo que cada mensagem seja exibida para a cor apropriada do alienígena.

5.6 – Estágios da vida: Escreva uma cadeia **`if-elif-else`** que determine o estágio da vida de uma pessoa. Defina um valor para a variável **`age`** e então:

- Se a pessoa tiver menos de 2 anos de idade, mostre uma mensagem dizendo que ela é um bebê.
- Se a pessoa tiver pelo menos 2 anos, mas menos de 4, mostre uma mensagem dizendo que ela é uma criança.
- Se a pessoa tiver pelo menos 4 anos, mas menos de 13, mostre uma mensagem

dizendo que ela é um(a) garoto(a).

- Se a pessoa tiver pelo menos 13 anos, mas menos de 20, mostre uma mensagem dizendo que ela é um(a) adolescente.
- Se a pessoa tiver pelo menos 20 anos, mas menos de 65, mostre uma mensagem dizendo que ela é adulto.
- Se a pessoa tiver 65 anos ou mais, mostre uma mensagem dizendo que essa pessoa é idoso.

5.7 – Fruta favorita: Faça uma lista de suas frutas favoritas e, então, escreva uma série de instruções `if` independentes que verifiquem se determinadas frutas estão em sua lista.

- Crie uma lista com suas três frutas favoritas e chame-a de `favorite_fruits`.
- Escreva cinco instruções `if`. Cada instrução deve verificar se uma determinada fruta está em sua lista. Se estiver, o bloco `if` deverá exibir uma frase, por exemplo, *Você realmente gosta de bananas!*

Usando instruções if com listas

Algumas tarefas interessantes podem ser feitas se combinarmos listas com instruções `if`. Podemos prestar atenção em valores especiais, que devam ser tratados de modo diferente de outros valores da lista. Podemos administrar mudanças de condições de modo eficiente, por exemplo, a disponibilidade de determinados itens em um restaurante durante um turno. Também podemos começar a provar que nosso código funciona conforme esperado em todas as possíveis situações.

Verificando itens especiais

Este capítulo começou com um exemplo simples mostrando como lidar com um valor especial como `'bmw'`, que devia ser exibido em um formato diferente de outros valores da lista. Agora que você tem uma compreensão básica a respeito dos testes condicionais e de instruções `if`, vamos observar com mais detalhes de que modo podemos prestar atenção em valores especiais de uma lista e tratá-los de forma apropriada.

Vamos prosseguir com o exemplo da pizzaria. A pizzaria exibe uma mensagem sempre que um ingrediente é adicionado à sua pizza à medida que ela é preparada. O código para essa ação pode ser escrito de modo

bem eficiente se criarmos uma lista de ingredientes solicitados pelo cliente e usarmos um laço para anunciar cada ingrediente à medida que ele é acrescentado à pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

A saída é simples, pois esse código é composto apenas de um laço `for` simples:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

E se a pizzaria ficasse sem pimentões verdes? Uma instrução `if` no laço `for` pode tratar essa situação de modo apropriado:

```
  requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

  for requested_topping in requested_toppings:
u     if requested_topping == 'green peppers':
          print("Sorry, we are out of green peppers right now.")
v     else:
          print("Adding " + requested_topping + ".")

  print("\nFinished making your pizza!")
```

Dessa vez verificamos cada item solicitado antes de adicioná-lo à pizza. O código em u verifica se a pessoa pediu pimentões verdes. Em caso afirmativo, exibimos uma mensagem informando por que ela não pode ter pimentões verdes. O bloco `else` em v garante que todos os demais ingredientes serão adicionados à pizza.

A saída mostra que cada ingrediente solicitado é tratado de forma

apropriada.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Verificando se uma lista não está vazia

Fizemos uma suposição simples sobre todas as listas com as quais trabalhamos até agora: supomos que toda lista continha pelo menos um item. Em breve, deixaremos os usuários fornecerem as informações a serem armazenadas em uma lista, portanto não poderemos supor que uma lista contenha um item sempre que um laço for executado. Nessa situação, será conveniente testar se uma lista está vazia antes de executar um laço `for`.

Como exemplo, vamos verificar se a lista de ingredientes solicitados está vazia antes de prepararmos a pizza. Se a lista estiver vazia, perguntaremos ao usuário se ele realmente quer uma pizza simples. Se a lista não estiver vazia, prepararemos a pizza exatamente como fizemos nos exemplos anteriores:

```
u requested_toppings = []

v if requested_toppings:
      for requested_topping in requested_toppings:
          print("Adding " + requested_topping + ".")
      print("\nFinished making your pizza!")
w else:
      print("Are you sure you want a plain pizza?")
```

Dessa vez, começamos com uma lista vazia de ingredientes solicitados em u. Em vez de passar diretamente para um laço `for`, fazemos uma verificação rápida em v. Quando o nome de uma lista é usado em uma instrução `if`, Python devolve `True` se a lista contiver pelo menos um item; uma lista vazia é avaliada como `False`. Se `requested_toppings` passar no teste condicional, executamos o mesmo laço `for` do exemplo anterior. Se o teste condicional falhar, exibimos uma mensagem perguntando ao cliente se ele realmente

quer uma pizza simples, sem ingredientes adicionais w.

Nesse caso, a lista está vazia, portanto a saída contém uma pergunta para saber se o usuário realmente quer uma pizza simples:

```
Are you sure you want a plain pizza?
```

Se a lista não estiver vazia, a saída mostrará cada ingrediente solicitado adicionado à pizza.

Usando várias listas

As pessoas pedirão de tudo, em especial quando se tratar de ingredientes para uma pizza. E se um cliente realmente quiser batatas fritas em sua pizza? Podemos usar listas e instruções `if` para garantir que o dado de entrada faça sentido antes de atuar sobre ele.

Vamos prestar atenção em solicitações de ingredientes incomuns antes de prepararmos uma pizza. O exemplo a seguir define duas listas. A primeira é uma lista de ingredientes disponíveis na pizzaria, e a segunda é a lista de ingredientes que o usuário pediu. Dessa vez, cada item em `requested_toppings` é verificado em relação à lista de ingredientes disponíveis antes de ser adicionado à pizza:

```
u available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

v requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

w for requested_topping in requested_toppings:
x     if requested_topping in available_toppings:
          print("Adding " + requested_topping + ".")
y     else:
          print("Sorry, we don't have " + requested_topping + ".")

  print("\nFinished making your pizza!")
```

Em u definimos uma lista de ingredientes disponíveis nessa pizzaria. Observe que essa informação poderia ser uma tupla se a pizzaria tiver uma seleção estável de ingredientes. Em v criamos uma lista de ingredientes solicitados por um cliente. Observe a solicitação incomum, `'french fries'`

(batatas fritas). Em `w` percorremos a lista de ingredientes solicitados em um laço. Nesse laço, inicialmente verificamos se cada ingrediente solicitado está na lista de ingredientes disponíveis `x`. Se estiver, adicionamos esse ingrediente na pizza. Se o ingrediente solicitado não estiver na lista de ingredientes disponíveis, o bloco `else` será executado `y`. Esse bloco exibe uma mensagem informando ao usuário quais ingredientes não estão disponíveis.

A sintaxe desse código gera uma saída clara e informativa:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

Com apenas algumas linhas de código, conseguimos tratar uma situação do mundo real de modo bem eficiente!

FAÇA VOCÊ MESMO

5.8 – Olá admin: Crie uma lista com cinco ou mais nomes de usuários, incluindo o nome `'admin'`. Suponha que você esteja escrevendo um código que exibirá uma saudação a cada usuário depois que eles fizerem login em um site. Percorra a lista com um laço e mostre uma saudação para cada usuário:

- Se o nome do usuário for `'admin'`, mostre uma saudação especial, por exemplo, *Olá admin, gostaria de ver um relatório de status?*
- Caso contrário, mostre uma saudação genérica, como *Olá Eric, obrigado por fazer login novamente*.

5.9 – Sem usuários: Acrescente um teste `if` em *hello_admin.py* para garantir que a lista de usuários não esteja vazia.

- Se a lista estiver vazia, mostre a mensagem *Precisamos encontrar alguns usuários!*
- Remova todos os nomes de usuário de sua lista e certifique-se de que a mensagem correta seja exibida.

5.10 – Verificando nomes de usuários: Faça o seguinte para criar um programa que simule o modo como os sites garantem que todos tenham um nome de usuário único.

- Crie uma lista chamada `current_users` com cinco ou mais nomes de usuários.

- Crie outra lista chamada `new_users` com cinco nomes de usuários. Garanta que um ou dois dos novos usuários também estejam na lista `current_users`.
- Percorra a lista `new_users` com um laço para ver se cada novo nome de usuário já foi usado. Em caso afirmativo, mostre uma mensagem informando que a pessoa deverá fornecer um novo nome. Se um nome de usuário não foi usado, apresente uma mensagem dizendo que o nome do usuário está disponível.
- Certifique-se de que sua comparação não levará em conta as diferenças entre letras maiúsculas e minúsculas. Se `'John'` foi usado, `'JOHN'` não deverá ser aceito.

5.11 – Números ordinais: Números ordinais indicam sua posição em uma lista, por exemplo, *1st* ou *2nd*, em inglês. A maioria dos números ordinais nessa língua termina com *th*, exceto 1, 2 e 3.

- Armazene os números de 1 a 9 em uma lista.
- Percorra a lista com um laço.
- Use uma cadeia **`if-elif-else`** no laço para exibir a terminação apropriada para cada número ordinal. Sua saída deverá conter `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, e cada resultado deve estar em uma linha separada.

Estilizando suas instruções if

Em todos os exemplos deste capítulo vimos bons hábitos de estilização. A única recomendação fornecida pela PEP 8 para estilizar testes condicionais é usar um único espaço em torno dos operadores de comparação, como `==`, `>=` e `<=`. Por exemplo:

```
if age < 4:
```

é melhor que:

```
if age<4:
```

Esse espaçamento não afeta o modo como Python interpreta seu código; ele simplesmente deixa seu código mais fácil de ler para você e para outras pessoas.

FAÇA VOCÊ MESMO

5.12 – Estilizando instruções `if`: Revise os programas que você escreveu neste capítulo e certifique-se de que os testes condicionais foram estilizados de forma

apropriada.

5.13 – Suas ideias: A essa altura, você é um programador mais capacitado do que era quando começou a ler este livro. Agora que você tem melhor noção de como situações do mundo real são modeladas em programas, talvez esteja pensando em alguns problemas que poderia resolver com seus próprios programas. Registre qualquer ideia nova que tiver sobre problemas que você queira resolver à medida que suas habilidades em programação continuam a melhorar. Considere jogos que você queira escrever, conjuntos de dados que possa querer explorar e aplicações web que gostaria de criar.

Resumo

Neste capítulo aprendemos a escrever testes condicionais, que são sempre avaliados como `True` ou `False`. Vimos como escrever instruções `if` simples, cadeias `if-else` e cadeias `if-elif-else`. Começamos a usar essas estruturas para identificar condições particulares que deviam ser testadas e aprendemos a reconhecer quando essas condições foram atendidas em nossos programas. Aprendemos a tratar determinados itens de uma lista de modo diferente de todos os demais itens, ao mesmo tempo que continuamos usando a eficiência do laço `for`. Também retomamos as recomendações de estilo de Python para garantir que seus programas cada vez mais complexos continuem relativamente fáceis de ler e de entender.

No Capítulo 6 conheceremos os dicionários de Python. Um dicionário é semelhante a uma lista, mas permite conectar informações. Aprenderemos a criar dicionários, percorrê-los com laços e usá-los em conjunto com listas e instruções `if`. Conhecer os dicionários permitirá modelar uma variedade ainda maior de situações do mundo real.

6
DICIONÁRIOS

Neste capítulo aprenderemos a usar dicionários Python, que permitem conectar informações relacionadas. Veremos como acessar as informações depois que elas estiverem em um dicionário e modificá-las. Pelo fato de os dicionários serem capazes de armazenar uma quantidade quase ilimitada de informações, mostraremos como percorrer os dados de um dicionário com um laço. Além disso, aprenderemos a aninhar dicionários em listas, listas em dicionários e até mesmo dicionários em outros dicionários.

Entender os dicionários permite modelar uma diversidade de objetos do mundo real de modo mais preciso. Você será capaz de criar um dicionário que representa uma pessoa e armazenar quantas informações quiser sobre ela. Poderá armazenar o nome, a idade, a localização, a profissão e qualquer outro aspecto de uma pessoa que possa ser descrito. Você será capaz de armazenar quaisquer duas informações que possam ser combinadas, por exemplo, uma lista de palavras e seus significados, uma lista de nomes de pessoas e seus números favoritos, uma lista de montanhas e suas altitudes, e assim por diante.

Um dicionário simples

Considere um jogo com alienígenas que possam ter cores e valores de pontuação diferentes. O dicionário simples a seguir armazena informações sobre um alienígena em particular:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
```

```
print(alien_0['points'])
```

O dicionário `alien_0` armazena a cor do alienígena e o valor da pontuação. As duas instruções `print` acessam e exibem essas informações, conforme mostrado a seguir:

```
green
5
```

Como ocorre com a maioria dos conceitos novos de programação, usar dicionários exige prática. Depois de trabalhar um pouco com dicionários, você logo verá a eficiência com que eles podem modelar situações do mundo real.

Trabalhando com dicionários

Um *dicionário* em Python é uma coleção de *pares chave-valor*. Cada *chave* é conectada a um valor, e você pode usar uma chave para acessar o valor associado a ela. O valor de uma chave pode ser um número, uma string, uma lista ou até mesmo outro dicionário. De fato, podemos usar qualquer objeto que possa ser criado em Python como valor de um dicionário.

Em Python, um dicionário é representado entre chaves, `{}`, com uma série de pares chave-valor entre elas, como mostramos no exemplo anterior:

```
alien_0 = {'color': 'green', 'points': 5}
```

Um *par chave-valor* é um conjunto de valores associados um ao outro. Quando fornecemos uma chave, Python devolve o valor associado a essa chave. Toda chave é associada a seu valor por meio de dois-pontos, e pares chave-valor individuais são separados por vírgulas. Podemos armazenar quantos pares chave-valor quisermos em um dicionário.

O dicionário mais simples tem exatamente um par chave-valor, como mostra esta versão modificada do dicionário `alien_0`:

```
alien_0 = {'color': 'green'}
```

Esse dicionário armazena uma informação sobre `alien_0`, que é a cor do alienígena. A string `'color'` é uma chave nesse dicionário, e seu valor associado é `'green'`.

Acessando valores em um dicionário

Para obter o valor associado a uma chave, especifique o nome do dicionário e coloque a chave entre colchetes, como vemos a seguir:

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

Essa instrução devolve o valor associado à chave `'color'` do dicionário `alien_0`:

```
green
```

Podemos ter um número ilimitado de pares chave-valor em um dicionário. Por exemplo, eis o dicionário `alien_0` original com dois pares chave-valor:

```
alien_0 = {'color': 'green', 'points': 5}
```

Agora podemos acessar a cor ou o valor da pontuação de `alien_0`. Se um jogador atingir esse alienígena, podemos consultar quantos pontos ele deve ganhar usando um código como este:

```
alien_0 = {'color': 'green', 'points': 5}

u new_points = alien_0['points']
v print("You just earned " + str(new_points) + " points!")
```

Depois que o dicionário for definido, o código em **u** extrai o valor associado à chave `'points'` do dicionário. Esse valor então é armazenado na variável `new_points`. A linha em **v** converte esse valor inteiro em uma string e exibe uma frase sobre quantos pontos o jogador acabou de ganhar:

```
You just earned 5 points!
```

Se executar esse código sempre que um alienígena for atingido, o valor da pontuação para esse alienígena será recuperado.

Adicionando novos pares chave-valor

Dicionários são estruturas dinâmicas, e você pode adicionar novos pares chave-valor a um dicionário a qualquer momento. Por exemplo, para

acrescentar um novo par chave-valor, especifique o nome do dicionário, seguido da nova chave entre colchetes, juntamente com o novo valor.

Vamos adicionar duas novas informações ao dicionário `alien_0`: as coordenadas x e y do alienígena, que nos ajudarão a exibir o alienígena em determinada posição da tela. Vamos colocar o alienígena na borda esquerda da tela, 25 pixels abaixo da margem superior. Como as coordenadas da tela normalmente começam no canto superior esquerdo da tela, posicionaremos o alienígena na margem esquerda definindo a coordenada x com 0, e 25 pixels abaixo da margem superior, definindo a coordenada y com o valor 25 positivo, como vemos aqui:

```
  alien_0 = {'color': 'green', 'points': 5}
  print(alien_0)

u alien_0['x_position'] = 0
v alien_0['y_position'] = 25
  print(alien_0)
```

Começamos definindo o mesmo dicionário com o qual estávamos trabalhando. Em seguida, exibimos esse dicionário, mostrando uma imagem de suas informações. Em u acrescentamos um novo par chave-valor ao dicionário: a chave `'x_position'` e o valor `0`. Fizemos o mesmo com a chave `'y_position'` em v. Ao exibir o dicionário modificado, vemos os dois pares chave-valor adicionais:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

A versão final do dicionário contém quatro pares chave-valor. Os dois pares originais especificam a cor e o valor da pontuação, enquanto os dois pares adicionais especificam a posição do alienígena. Observe que a ordem dos pares chave-valor não coincide com a ordem em que os adicionamos. Python não se importa com a ordem em que armazenamos cada par chave-valor; ele só se importa com a conexão entre cada chave e seu valor.

Começando com um dicionário vazio

Às vezes, é conveniente ou até mesmo necessário começar com um

dicionário vazio e então acrescentar novos itens a ele. Para começar a preencher um dicionário vazio, defina-o com um conjunto de chaves vazio e depois acrescente cada par chave-valor em sua própria linha. Por exemplo, eis o modo de criar o dicionário `alien_0` usando esta abordagem:

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Nesse caso, definimos um dicionário `alien_0` vazio e, em seguida, adicionamos valores para cor e pontuação. O resultado é o dicionário que usamos nos exemplos anteriores:

```
{'color': 'green', 'points': 5}
```

Geralmente usamos dicionários vazios quando armazenamos dados fornecidos pelo usuário em um dicionário, ou quando escrevemos um código que gere um grande número de pares chave-valor automaticamente.

Modificando valores em um dicionário

Para modificar um valor em um dicionário, especifique o nome do dicionário com a chave entre colchetes e o novo valor que você quer associar a essa chave. Por exemplo, considere um alienígena que muda de verde para amarelo à medida que o jogo prossegue:

```
alien_0 = {'color': 'green'}
print("The alien is " + alien_0['color'] + ".")

alien_0['color'] = 'yellow'
print("The alien is now " + alien_0['color'] + ".")
```

Inicialmente, definimos um dicionário para `alien_0` que contém apenas a cor do alienígena; em seguida, modificamos o valor associado à chave `'color'` para `'yellow'`. A saída mostra que o alienígena realmente mudou de verde para amarelo:

```
The alien is green.
The alien is now yellow.
```

Para ver um exemplo mais interessante, vamos monitorar a posição de um alienígena que pode se deslocar com velocidades diferentes. Armazenaremos um valor que representa a velocidade atual do alienígena e, então, usaremos esse valor para determinar a distância que o alienígena deve se mover para a direita:

```
  alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
  print("Original x-position: " + str(alien_0['x_position']))

  # Move o alienígena para a direita
  # Determina a distância que o alienígena deve se deslocar de acordo com
  sua
  # velocidade atual
u if alien_0['speed'] == 'slow':
      x_increment = 1
  elif alien_0['speed'] == 'medium':
      x_increment = 2
  else:
      # Este deve ser um alienígena rápido
      x_increment = 3

  # A nova posição é a posição antiga somada ao incremento
v alien_0['x_position'] = alien_0['x_position'] + x_increment

  print("New x-position: " + str(alien_0['x_position']))
```

Começamos definindo um alienígena com uma posição *x* e *y* iniciais, e uma velocidade igual a `'medium'`. Omitimos a cor e os valores da pontuação por questões de simplicidade, mas esse exemplo funcionaria da mesma maneira se tivéssemos incluído esses pares chave-valor também. Além disso, exibimos o valor original de `x_position` para ver a distância com que o alienígena se desloca para a direita.

Em u uma cadeia `if-elif-else` determina a distância que o alienígena deve se mover para a direita e esse valor é armazenado na variável `x_increment`. Se a velocidade do alienígena for `'slow'`, ele se deslocará de uma unidade para a direita; se a velocidade for `'medium'`, se deslocará de duas unidades

para a direita e, se for 'fast', se deslocará de três unidades à direita. Depois de calculado, o incremento é somado ao valor de x_position em v, e o resultado é armazenado no x_position do dicionário.

Como esse é um alienígena de velocidade média, sua posição se desloca de duas unidades para a direita:

```
Original x-position: 0
New x-position: 2
```

Essa técnica é bem interessante: ao mudar um valor do dicionário do alienígena, podemos mudar o seu comportamento como um todo. Por exemplo, para transformar esse alienígena de velocidade média em um alienígena rápido, acrescente a linha a seguir:

```
alien_0['speed'] = fast
```

O bloco if-elif-else então atribuirá um valor maior para x_increment na próxima vez que o código for executado.

Removendo pares chave-valor

Quando não houver mais necessidade de uma informação armazenada em um dicionário, podemos usar a instrução del para remover totalmente um par chave-valor. Tudo de que del precisa é do nome do dicionário e da chave que você deseja remover.

Por exemplo, vamos remover a chave 'points' do dicionário alien_0, juntamente com seu valor:

```
  alien_0 = {'color': 'green', 'points': 5}
  print(alien_0)

u del alien_0['points']
  print(alien_0)
```

A linha em u diz a Python para apagar a chave 'points' do dicionário alien_0 e remover o valor associado a essa chave também. A saída mostra que a chave 'points' e seu valor igual a 5 foram apagados, porém o restante do dicionário não foi afetado:

```
{'color': 'green', 'points': 5}
```

```
{'color': 'green'}
```

NOTA Saiba que o par chave-valor apagado é removido de forma permanente.

Um dicionário de objetos semelhantes

O exemplo anterior envolveu a armazenagem de diferentes tipos de informação sobre um objeto: um alienígena em um jogo. Também podemos usar um dicionário para armazenar um tipo de informação sobre vários objetos. Por exemplo, suponha que você queira fazer uma enquete com várias pessoas e perguntar-lhes qual é a sua linguagem de programação favorita. Um dicionário é conveniente para armazenar os resultados de uma enquete simples, desta maneira:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }
```

Como podemos ver, dividimos um dicionário maior em várias linhas. Cada chave é o nome de uma pessoa que respondeu à enquete, e cada valor é a sua opção de linguagem. Quando souber que precisará de mais de uma linha para definir um dicionário, tecle ENTER depois da chave de abertura. Em seguida, indente a próxima linha em um nível (quatro espaços) e escreva o primeiro par chave-valor, seguido de uma vírgula. A partir desse ponto, quando pressionar ENTER, seu editor de texto deverá indentar automaticamente todos os pares chave-valor subsequentes para que estejam de acordo com o primeiro par chave-valor.

Depois que acabar de definir o dicionário, acrescente uma chave de fechamento em uma nova linha após o último par chave-valor e indente-a em um nível para que esteja alinhada com as chaves do dicionário. Incluir uma vírgula após o último par chave-valor também é uma boa prática; assim você estará preparado para acrescentar um novo par chave-valor na próxima linha.

NOTA A maioria dos editores tem alguma funcionalidade que ajuda a formatar listas e dicionários longos, de modo semelhante a esse exemplo. Outras maneiras

aceitáveis de formatar dicionários longos também estão disponíveis, portanto você poderá ver formatações um pouco diferentes em seu editor ou em outros códigos-fonte.

Para usar esse dicionário, dado o nome de uma pessoa que tenha respondido à enquete, podemos consultar facilmente sua linguagem favorita:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

u print("Sarah's favorite language is " +
v     favorite_languages['sarah'].title() +
w     ".")
```

Para ver a linguagem que Sarah escolheu, solicitamos o valor em:

```
favorite_languages['sarah']
```

Essa sintaxe é usada na instrução `print` em v, e a saída mostra a linguagem predileta de Sarah:

```
Sarah's favorite language is C.
```

Esse exemplo também mostra como podemos dividir uma instrução `print` longa em várias linhas. A palavra `print` é menor que a maioria dos nomes de dicionário, portanto faz sentido incluir a primeira parte do que você quer exibir logo depois do parêntese de abertura u. Escolha um ponto apropriado para separar o que está sendo exibido e acrescente um operador de concatenação (+) no final da primeira linha v. Tecle ENTER e depois TAB para alinhar todas as linhas subsequentes em um nível de indentação abaixo da instrução `print`. Quando terminar de compor sua saída, você pode colocar o parêntese de fechamento na última linha do bloco `print` w.

FAÇA VOCÊ MESMO

6.1 – Pessoa: Use um dicionário para armazenar informações sobre uma pessoa que você conheça. Armazene seu primeiro nome, o sobrenome, a idade e a cidade em que ela vive. Você deverá ter chaves como `first_name`, `last_name`, `age` e `city`. Mostre cada informação armazenada em seu dicionário.

6.2 – Números favoritos: Use um dicionário para armazenar os números favoritos de algumas pessoas. Pense em cinco nomes e use-os como chaves em seu dicionário. Pense em um número favorito para cada pessoa e armazene cada um como um valor em seu dicionário. Exiba o nome de cada pessoa e seu número favorito. Para que seja mais divertido ainda, faça uma enquete com alguns amigos e obtenha alguns dados reais para o seu programa.

6.3 – Glossário: Um dicionário Python pode ser usado para modelar um dicionário de verdade. No entanto, para evitar confusão, vamos chamá-lo de glossário.

- Pense em cinco palavras relacionadas à programação que você conheceu nos capítulos anteriores. Use essas palavras como chaves em seu glossário e armazene seus significados como valores.
- Mostre cada palavra e seu significado em uma saída formatada de modo elegante. Você pode exibir a palavra seguida de dois-pontos e depois o seu significado, ou apresentar a palavra em uma linha e então exibir seu significado indentado em uma segunda linha. Utilize o caractere de quebra de linha (`\n`) para inserir uma linha em branco entre cada par palavra-significado em sua saída.

Percorrendo um dicionário com um laço

Um único dicionário Python pode conter apenas alguns pares chave-valor ou milhões deles. Como um dicionário pode conter uma grande quantidade de dados, Python permite percorrer um dicionário com um laço. Dicionários podem ser usados para armazenar informações de várias maneiras; assim, há diversos modos diferentes de percorrê-los com um laço. Podemos percorrer todos os pares chave-valor de um dicionário usando suas chaves ou seus valores.

Percorrendo todos os pares chave-valor com um laço

Antes de explorar as diferentes abordagens para percorrer um dicionário com um laço, vamos considerar um novo dicionário projetado para armazenar informações sobre um usuário em um site. O dicionário a seguir

armazenará o nome de usuário, o primeiro nome e o sobrenome de uma pessoa:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

Podemos acessar qualquer informação única sobre `user_0` com base no que já aprendemos neste capítulo. E se quiséssemos ver tudo que está armazenado no dicionário desse usuário? Para isso, podemos percorrer o dicionário com um laço `for`:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

u for key, value in user_0.items():
v     print("\nKey: " + key)
w     print("Value: " + value)
```

Como vemos em u, para escrever um laço `for` para um dicionário, devemos criar nomes para as duas variáveis que armazenarão a chave e o valor de cada par chave-valor. Você pode escolher qualquer nome que quiser para essas duas variáveis. Esse código também funcionaria bem se usássemos abreviaturas para os nomes das variáveis, assim:

```
for k, v in user_0.items()
```

A segunda metade da instrução `for` em u inclui o nome do dicionário, seguido do método `items()`, que devolve uma lista de pares chave-valor. O laço `for` então armazena cada um desses pares nas duas variáveis especificadas. No exemplo anterior, usamos as variáveis para exibir cada chave (`key`) v, seguido do valor associado (`value`) w. O `"\n"` na primeira instrução `print` garante que uma linha em branco seja inserida antes de

cada par chave-valor na saída:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Observe novamente que os pares chave-valor não são devolvidos na ordem em que foram armazenados, mesmo quando percorremos o dicionário com um laço. Python não se importa com a ordem em que os pares chave-valor são armazenados; ele só registra as conexões entre cada chave individual e seu valor.

Percorrer todos os pares chave-valor com um laço funciona bem, em particular, para dicionários como o do exemplo em *favorite_languages.py*, que armazena o mesmo tipo de informação para várias chaves diferentes. Se percorrermos o dicionário `favorite_languages` com um laço, teremos o nome de cada pessoa no dicionário e sua linguagem de programação favorita. Como as chaves sempre se referem ao nome de uma pessoa e o valor é sempre uma linguagem, usaremos as variáveis `name` e `language` no laço, em vez de `key` e `value`. Isso facilita entender o que está acontecendo:

favorite_languages.py

```
  favorite_languages = {
      'jen': 'python',
      'sarah': 'c',
      'edward': 'ruby',
      'phil': 'python',
      }

u for name, language in favorite_languages.items():
v     print(name.title() + "'s favorite language is " +
          language.title() + ".")
```

O código em u diz a Python para percorrer todos os pares chave-valor do dicionário com um laço. À medida que cada par é tratado, a chave é

armazenada na variável `name` e o valor, na variável `language`. Esses nomes descritivos fazem com que seja muito mais fácil ver o que a instrução `print` em v faz.

Agora, com apenas algumas linhas de código, podemos exibir todas as informações da enquete:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Phil's favorite language is Python.
Edward's favorite language is Ruby.
```

Esse tipo de laço funcionaria igualmente bem se nosso dicionário armazenasse os resultados da enquete para mil ou até mesmo para um milhão de pessoas.

Percorrendo todas as chaves de um dicionário com um laço

O método `keys()` é conveniente quando não precisamos trabalhar com todos os valores de um dicionário. Vamos percorrer o dicionário `favorite_languages` com um laço e exibir os nomes de todos que responderam à enquete:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

u for name in favorite_languages.keys():
    print(name.title())
```

A linha em u diz a Python para extrair todas as chaves do dicionário `favorite_languages` e armazená-las, uma de cada vez, na variável `name`. A saída mostra os nomes de todos que participaram da enquete:

```
Jen
Sarah
Phil
Edward
```

Percorrer as chaves, na verdade, é o comportamento-padrão quando percorremos um dicionário com um laço, portanto este código produzirá a mesma saída se escrevêssemos:

```
for name in favorite_languages:
```

em vez de:

```
for name in favorite_languages.keys():
```

Você pode optar por usar o método `keys()` explicitamente se isso deixar seu código mais fácil de ler, ou pode omiti-lo, se quiser.

Podemos acessar o valor associado a qualquer chave que nos interessar no laço usando a chave atual. Vamos exibir uma mensagem a dois amigos sobre as linguagens que eles escolheram. Percorreremos os nomes no dicionário com um laço como fizemos antes, porém, quando o nome for de um de nossos amigos, apresentaremos uma mensagem sobre sua linguagem favorita:

```
 favorite_languages = {
     'jen': 'python',
     'sarah': 'c',
     'edward': 'ruby',
     'phil': 'python',
     }

u friends = ['phil', 'sarah']
 for name in favorite_languages.keys():
     print(name.title())

v     if name in friends:
         print("  Hi " + name.title() +
             ", I see your favorite language is " +
w            favorite_languages[name].title() + "!")
```

Em u criamos uma lista de amigos a quem queremos exibir uma mensagem. No laço, exibimos o nome de cada pessoa. Então, em v, verificamos se o nome com que estamos trabalhando (`name`) está na lista `friends`. Se estiver, exibimos uma saudação especial, incluindo uma referência à sua opção de linguagem. Para acessar a linguagem favorita em

w, usamos o nome do dicionário e o valor atual de `name` como chave. O nome de todas as pessoas é exibido, porém nossos amigos recebem uma mensagem especial:

```
Edward
Phil
  Hi Phil, I see your favorite language is Python!
Sarah
  Hi Sarah, I see your favorite language is C!
Jen
```

Você também pode usar o método `keys()` para descobrir se uma pessoa em particular participou da enquete. Dessa vez, vamos ver se Erin respondeu à enquete:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

u if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

O método `keys()` não serve apenas para laços: na verdade, ele devolve uma lista de todas as chaves, e a linha em u simplesmente verifica se `'erin'` está nessa lista. Como ela não está, uma mensagem é exibida convidando-a a participar da enquete:

```
Erin, please take our poll!
```

Percorrendo as chaves de um dicionário em ordem usando um laço

Um dicionário sempre mantém uma conexão clara entre cada chave e seu valor associado, mas você não obterá os itens de um dicionário em uma ordem previsível. Isso não é um problema, pois, geralmente, queremos apenas obter o valor correto associado a cada chave.

Uma maneira de fazer os itens serem devolvidos em determinada sequência é ordenar as chaves à medida que são devolvidas no laço `for`.

Podemos usar a função `sorted()` para obter uma cópia ordenada das chaves:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(name.title() + ", thank you for taking the poll.")
```

Essa instrução `for` é como as outras instruções `for`, exceto que a função `sorted()` está em torno do método `dictionary.keys()`. Isso diz a Python para listar todas as chaves do dicionário e ordenar essa lista antes de percorrê-la com um laço. A saída mostra os nomes de todos que responderam à enquete, exibidos de forma ordenada:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Percorrendo todos os valores de um dicionário com um laço

Se você estiver mais interessado nos valores contidos em um dicionário, o método `values()` pode ser usado para devolver uma lista de valores sem as chaves. Por exemplo, suponha que queremos apenas uma lista de todas as linguagens escolhidas em nossa enquete sobre linguagens de programação, sem o nome da pessoa que escolheu cada linguagem:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
```

```
for language in favorite_languages.values():
    print(language.title())
```

A instrução `for`, nesse caso, extrai cada valor do dicionário e o armazena na variável `language`. Quando esses valores são exibidos, temos uma lista de todas as linguagens escolhidas:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

Essa abordagem extrai todos os valores do dicionário, sem verificar se há repetições. Isso pode funcionar bem com uma quantidade pequena de valores, mas em uma enquete com um número grande de entrevistados, o resultado seria uma lista com muitas repetições. Para ver cada linguagem escolhida sem repetições, podemos usar um *conjunto* (set). Um conjunto é semelhante a uma lista, exceto que cada item de um conjunto deve ser único:

```
  favorite_languages = {
      'jen': 'python',
      'sarah': 'c',
      'edward': 'ruby',
      'phil': 'python',
      }

  print("The following languages have been mentioned:")
u for language in set(favorite_languages.values()):
      print(language.title())
```

Quando colocamos `set()` em torno de uma lista que contenha itens duplicados, Python identifica os itens únicos na lista e cria um conjunto a partir desses itens. Em **u** usamos `set()` para extrair as linguagens únicas em `favorite_languages.values()`.

O resultado é uma lista de linguagens mencionadas pelas pessoas que participaram da enquete, sem repetições:

```
The following languages have been mentioned:
```

```
Python
C
Ruby
```

À medida que continuar a conhecer Python, com frequência você encontrará um recurso embutido na linguagem que o ajudará a fazer exatamente o que quiser com seus dados.

FAÇA VOCÊ MESMO

6.4 – Glossário 2: Agora que você já sabe como percorrer um dicionário com um laço, limpe o código do Exercício 6.3 (página 148), substituindo sua sequência de instruções `print` por um laço que percorra as chaves e os valores do dicionário. Quando tiver certeza de que seu laço funciona, acrescente mais cinco termos de Python ao seu glossário. Ao executar seu programa novamente, essas palavras e significados novos deverão ser automaticamente incluídos na saída.

6.5 – Rios: Crie um dicionário que contenha três rios importantes e o país que cada rio corta. Um par chave-valor poderia ser `'nilo': 'egito'`.

- Use um laço para exibir uma frase sobre cada rio, por exemplo, *O Nilo corre pelo Egito*.
- Use um laço para exibir o nome de cada rio incluído no dicionário.
- Use um laço para exibir o nome de cada país incluído no dicionário.

6.6 – Enquete: Utilize o código em *favorite_languages.py* (página 150).

- Crie uma lista de pessoas que devam participar da enquete sobre linguagem favorita. Inclua alguns nomes que já estejam no dicionário e outros que não estão.
- Percorra a lista de pessoas que devem participar da enquete. Se elas já tiverem respondido à enquete, mostre uma mensagem agradecendo-lhes por responder. Se ainda não participaram da enquete, apresente uma mensagem convidando-as a responder.

Informações aninhadas

Às vezes, você vai querer armazenar um conjunto de dicionários em uma lista ou uma lista de itens como um valor em um dicionário. Isso é conhecido como *aninhar* informações. Podemos aninhar um conjunto de dicionários em uma lista, uma lista de itens em um dicionário ou até mesmo um dicionário em outro dicionário. Aninhar informações é um

recurso eficaz, como mostrarão os próximos exemplos.

Uma lista de dicionários

O dicionário `alien_0` contém várias informações sobre um alienígena, mas não há espaço para armazenar informações sobre um segundo alienígena, muito menos para uma tela cheia deles. Como podemos administrar uma frota de alienígenas? Uma maneira é criar uma lista de alienígenas, em que cada alienígena seja representado por um dicionário com informações sobre ele. Por exemplo, o código a seguir cria uma lista com três alienígenas:

aliens.py

```
  alien_0 = {'color': 'green', 'points': 5}
  alien_1 = {'color': 'yellow', 'points': 10}
  alien_2 = {'color': 'red', 'points': 15}

u aliens = [alien_0, alien_1, alien_2]

  for alien in aliens:
      print(alien)
```

Inicialmente criamos três dicionários, cada um representando um alienígena diferente. Em u reunimos esses dicionários em uma lista chamada `aliens`. Por fim, percorremos a lista com um laço e exibimos cada alienígena:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Um exemplo mais realista envolveria mais de três alienígenas, com um código que gere automaticamente cada alienígena. No exemplo a seguir, usamos `range()` para criar uma frota de 30 alienígenas:

```
  # Cria uma lista vazia para armazenar alienígenas
  aliens = []

  # Cria 30 alienígenas verdes
u for alien_number in range(30):
```

```
v     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
w     aliens.append(new_alien)

  # Mostra os 5 primeiros alienígenas
x for alien in aliens[:5]:
      print(alien)
  print("...")

  # Mostra quantos alienígenas foram criados
y print("Total number of aliens: " + str(len(aliens)))
```

Esse exemplo começa com uma lista vazia para armazenar todos os alienígenas que serão criados. Em u `range()` devolve um conjunto de números, que simplesmente diz a Python quantas vezes queremos que o laço se repita. A cada vez que o laço é executado, criamos um novo alienígena v e concatenamos cada novo alienígena à lista `aliens` w. Em x usamos uma fatia para exibir os cinco primeiros alienígenas e, em seguida, em y, exibimos o tamanho da lista para provar que realmente geramos a frota completa com 30 alienígenas:

```
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...

Total number of aliens: 30
```

Todos esses alienígenas têm as mesmas características, mas Python considera cada um como um objeto diferente, o que nos permite modificar cada alienígena individualmente.

Como podemos trabalhar com um conjunto de alienígenas como esse? Suponha que um aspecto do jogo consista em fazer alguns alienígenas mudarem de cor e moverem-se mais rápido à medida que o jogo prosseguir. Quando for o momento de mudar de cor, podemos usar um laço `for` e uma instrução `if` para alterar a cor dos alienígenas. Por exemplo, para mudar os três primeiros alienígenas para amarelo, com velocidade

média, valendo dez pontos cada, podemos fazer o seguinte:

```
# Cria uma lista vazia para armazenar alienígenas
aliens = []

# Cria 30 alienígenas verdes
for alien_number in range (0,30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Mostra os 5 primeiros alienígenas
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Como queremos modificar os três primeiros alienígenas, percorremos uma fatia que inclui apenas os três primeiros alienígenas. Todos os alienígenas são verdes agora, mas isso nem sempre ocorrerá, portanto escrevemos uma instrução `if` para garantir que estamos modificando apenas os alienígenas verdes. Se o alienígena for verde, mudamos a cor para `'yellow'`, a velocidade para `'medium'` e o valor da pontuação para `10`, como vemos na saída a seguir:

```
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...
```

Poderíamos expandir esse laço acrescentando um bloco `elif` que transforme alienígenas amarelos em alienígenas vermelhos e rápidos, que valem 15 pontos cada. Sem mostrar o programa todo novamente, esse laço teria o seguinte aspecto:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

É comum armazenar vários dicionários em uma lista quando cada dicionário tiver diversos tipos de informação sobre um objeto. Por exemplo, podemos criar um dicionário para cada usuário de um site, como fizemos em *user.py*, e armazenar os dicionários individuais em uma lista chamada `users`. Todos os dicionários da lista devem ter uma estrutura idêntica para que possamos percorrer a lista com um laço e trabalhar com cada objeto representado por um dicionário do mesmo modo.

Uma lista em um dicionário

Em vez de colocar um dicionário em uma lista, às vezes é conveniente colocar uma lista em um dicionário. Por exemplo, considere como podemos descrever uma pizza que alguém está pedindo. Se usássemos apenas uma lista, tudo que poderíamos realmente armazenar é uma lista dos ingredientes da pizza. Com um dicionário, uma lista de ingredientes pode ser apenas um dos aspectos da pizza que estamos descrevendo.

No exemplo a seguir, dois tipos de informação são armazenados para cada pizza: o tipo de massa e a lista de ingredientes. A lista de ingredientes é um valor associado à chave `'toppings'`. Para usar os itens da lista, fornecemos o nome do dicionário e a chave `'toppings'`, como faríamos com qualquer valor do dicionário. Em vez de devolver um único valor, teremos uma lista de ingredientes:

pizza.py

```
  # Armazena informações sobre uma pizza que está sendo pedida
u pizza = {
      'crust': 'thick',
```

```
    'toppings': ['mushrooms', 'extra cheese'],
    }

  # Resume o pedido
v print("You ordered a " + pizza['crust'] + "-crust pizza " +
    "with the following toppings:")

w for topping in pizza['toppings']:
    print("\t" + topping)
```

Começamos em u com um dicionário que armazena informações sobre uma pizza que está sendo pedida. Uma das chaves do dicionário é `'crust'`, e o valor associado é a string `'thick'`. A próxima chave, `'toppings'`, tem como valor uma lista que armazena todos os ingredientes solicitados. Em v resumimos o pedido antes de preparar a pizza. Para exibir os ingredientes, escrevemos um laço `for` w. Para acessar a lista de ingredientes, usamos a chave `'toppings'`, e Python obtém a lista de ingredientes do dicionário.

A saída a seguir oferece um resumo da pizza que planejamos preparar:

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

Você pode aninhar uma lista em um dicionário sempre que quiser que mais de um valor seja associado a uma única chave em um dicionário. No exemplo anterior das linguagens de programação favoritas, se armazenássemos as respostas de cada pessoa em uma lista, elas poderiam escolher mais de uma linguagem predileta. Se percorrermos o dicionário com um laço, o valor associado a cada pessoa será uma lista das linguagens, e não uma única linguagem. No laço `for` do dicionário, usamos outro laço `for` para percorrer a lista de linguagens associada a cada pessoa:

favorite_languages.py

```
u favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
    }
```

```
v for name, languages in favorite_languages.items():
     print("\n" + name.title() + "'s favorite languages are:")
w     for language in languages:
          print("\t" + language.title())
```

Como podemos ver em u, o valor associado a cada nome agora é uma lista. Observe que algumas pessoas têm uma linguagem favorita, enquanto outras têm várias. Quando percorremos o dicionário em v com um laço, usamos uma variável de nome `languages` para armazenar cada valor do dicionário, pois sabemos que esse valor será uma lista. No laço principal do dicionário, usamos outro laço `for` w para percorrer a lista de linguagens favoritas de cada pessoa. Agora cada pessoa pode listar quantas linguagens favoritas quiser:

```
Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Phil's favorite languages are:
    Python
    Haskell

Edward's favorite languages are:
    Ruby
    Go
```

Para aperfeiçoar mais ainda esse programa, podemos incluir uma instrução `if` no início do laço `for` do dicionário para ver se cada pessoa tem mais de uma linguagem favorita analisando o valor de `len(languages)`. Se uma pessoa tiver mais de uma linguagem favorita, a saída será a mesma. Se ela tiver apenas uma linguagem predileta, poderíamos mudar a frase para refletir esse fato. Por exemplo, você poderia dizer: `Sarah's favorite language is C`.

NOTA Não aninhe listas e dicionários com muitos níveis de profundidade. Se estiver aninhando itens com um nível de profundidade muito maior do que vimos nos

exemplos anteriores ou se estiver trabalhando com o código de outra pessoa, e esse código tiver níveis significativos de informações aninhadas, é mais provável que haja uma maneira mais simples de solucionar o problema existente.

Um dicionário em um dicionário

Podemos aninhar um dicionário em outro dicionário, mas o código poderá ficar complicado rapidamente se isso for feito. Por exemplo, se você tiver vários usuários em um site, cada um com um nome único, os nomes dos usuários poderão ser usados como chaves em um dicionário. Você poderá então armazenar informações sobre cada usuário usando um dicionário como o valor associado a cada nome de usuário. Na listagem a seguir, armazenamos três informações sobre cada usuário: seu primeiro nome, o sobrenome e a localidade. Acessaremos essas informações percorrendo os nomes dos usuários em um laço e o dicionário de informações associado a cada nome de usuário:

many_users.py

```
  users = {
      'aeinstein': {
          'first': 'albert',
          'last': 'einstein',
          'location': 'princeton',
          },

      'mcurie': {
          'first': 'marie',
          'last': 'curie',
          'location': 'paris',
          },

      }

u for username, user_info in users.items():
v     print("\nUsername: " + username)
w      full_name = user_info['first'] + " " + user_info['last']
      location = user_info['location']

x     print("\tFull name: " + full_name.title())
```

```
    print("\tLocation: " + location.title())
```

Inicialmente definimos um dicionário chamado `users` com duas chaves, uma para cada um dos seguintes nomes de usuário: `'aeinstein'` e `'mcurie'`. O valor associado a cada chave é um dicionário que inclui o primeiro nome, o sobrenome e a localidade de cada usuário. Em **u** percorremos o dicionário `users` com um laço. Python armazena cada chave na variável `username` e o dicionário associado a cada nome de usuário na variável `user_info`. Depois que entrarmos no laço principal do dicionário, exibimos o nome do usuário em **v**.

Em **w** começamos a acessar o dicionário interno. A variável `user_info`, que contém o dicionário com informações do usuário, tem três chaves: `'first'`, `'last'` e `'location'`. Usamos cada chave para gerar um nome completo e a localidade formatados de modo elegante para cada pessoa **x**:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton

Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Observe que a estrutura do dicionário de cada usuário é idêntica. Embora Python não exija, essa estrutura facilita trabalhar com dicionários aninhados. Se o dicionário de cada pessoa tivesse chaves diferentes, o código no laço `for` seria mais complicado.

FAÇA VOCÊ MESMO

6.7 – Pessoas: Comece com o programa que você escreveu no Exercício 6.1 (página 147). Crie dois novos dicionários que representem pessoas diferentes e armazene os três dicionários em uma lista chamada `people`. Percorra sua lista de pessoas com um laço. À medida que percorrer a lista, apresente tudo que você sabe sobre cada pessoa.

6.8 – Animais de estimação: Crie vários dicionários, em que o nome de cada dicionário seja o nome de um animal de estimação. Em cada dicionário, inclua o tipo do animal e o nome do dono. Armazene esses dicionários em uma lista chamada `pets`. Em seguida, percorra sua lista com um laço e, à medida que fizer

isso, apresente tudo que você sabe sobre cada animal de estimação.

6.9 – Lugares favoritos: Crie um dicionário chamado `favorite_places`. Pense em três nomes para usar como chaves do dicionário e armazene de um a três lugares favoritos para cada pessoa. Para deixar este exercício um pouco mais interessante, peça a alguns amigos que nomeiem alguns de seus lugares favoritos. Percorra o dicionário com um laço e apresente o nome de cada pessoa e seus lugares favoritos.

6.10 – Números favoritos: Modifique o seu programa do Exercício 6.2 (página 147) para que cada pessoa possa ter mais de um número favorito. Em seguida, apresente o nome de cada pessoa, juntamente com seus números favoritos.

6.11 – Cidades: Crie um dicionário chamado `cities`. Use os nomes de três cidades como chaves em seu dicionário. Crie um dicionário com informações sobre cada cidade e inclua o país em que a cidade está localizada, a população aproximada e um fato sobre essa cidade. As chaves do dicionário de cada cidade devem ser algo como `country`, `population` e `fact`. Apresente o nome de cada cidade e todas as informações que você armazenou sobre ela.

6.12 – Extensões: Estamos trabalhando agora com exemplos complexos o bastante para poderem ser estendidos de várias maneiras. Use um dos programas de exemplo deste capítulo e estenda-o acrescentando novas chaves e valores, alterando o contexto do programa ou melhorando a formatação da saída.

Resumo

Neste capítulo aprendemos a definir um dicionário e a trabalhar com as informações armazenadas nele. Vimos como acessar e modificar elementos individuais de um dicionário e percorrer todas as suas informações com um laço. Aprendemos a percorrer todos os pares chave-valor de um dicionário, suas chaves e seus valores com um laço. Também vimos como aninhar vários dicionários em uma lista, aninhar listas em um dicionário e um dicionário em outro dicionário.

No próximo capítulo, conheceremos os laços `while` e veremos como aceitar dados de entrada das pessoas que usarem seus programas. Será um capítulo empolgante, pois você aprenderá a deixar todos os seus programas interativos: eles serão capazes de responder a dados de entrada do usuário.

7
ENTRADA DE USUÁRIO E LAÇOS WHILE

A maioria dos programas é escrita para resolver o problema de um usuário final. Para isso, geralmente precisamos obter algumas informações do usuário. Como exemplo simples, vamos supor que uma pessoa queira descobrir se ela tem idade suficiente para votar. Se você escrever um programa que responda a essa pergunta, será necessário saber a idade do usuário antes de poder oferecer uma resposta. O programa precisará pedir ao usuário que forneça a idade, ou *entre* com ela; depois que tiver esse dado de entrada, o programa poderá comparar esse valor com a idade para votar a fim de determinar se o usuário tem idade suficiente e então informar o resultado.

Neste capítulo aprenderemos a aceitar dados de entrada do usuário para que seu programa possa então trabalhar com eles. Quando seu programa precisar de um nome, você poderá solicitá-lo ao usuário. Quando precisar de uma lista de nomes, você poderá pedir uma série de nomes ao usuário. Para isso, usaremos a função `input()`.

Também veremos como manter os programas executando pelo tempo que os usuários quiserem, de modo que eles possam fornecer quantas informações forem necessárias; em seguida, seu programa poderá trabalhar com essas informações. Usaremos o laço `while` de Python para manter os programas executando enquanto determinadas condições permanecerem verdadeiras.

Com a capacidade de trabalhar com dados de entrada do usuário e controlar o tempo que seus programas executam, você poderá escrever programas totalmente interativos.

Como a função input() trabalha

A função `input()` faz uma pausa em seu programa e espera o usuário fornecer um texto. Depois que Python recebe a entrada do usuário, esse dado é armazenado em uma variável para que você possa trabalhar com ele de forma conveniente.

Por exemplo, o programa a seguir pede que o usuário forneça um texto e, em seguida, exibe essa mensagem de volta ao usuário:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you:
")
print(message)
```

A função `input()` aceita um argumento: o *prompt* – ou as instruções – que queremos exibir ao usuário para que eles saibam o que devem fazer. Nesse exemplo, quando Python executar a primeira linha, o usuário verá o prompt `Tell me something, and I will repeat it back to you: `. O programa espera enquanto o usuário fornece sua resposta e continua depois que ele tecla ENTER. A resposta é armazenada na variável `message`; depois disso, `print(message)` exibe a entrada de volta ao usuário:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTA O Sublime Text não executa programas que pedem uma entrada ao usuário. Você pode usar o Sublime Text para escrever programas que solicitem uma entrada, mas será necessário executar esses programas a partir de um terminal. Consulte a seção “Executando programas Python a partir de um terminal”.

Escrevendo prompts claros

Sempre que usar a função `input()`, inclua um prompt claro, fácil de compreender, que informe o usuário exatamente que tipo de informação você procura. Qualquer frase que diga aos usuários o que eles devem fornecer será apropriada. Por exemplo:

greeter.py

```
name = input("Please enter your name: ")
print("Hello, " + name + "!")
```

Acrescente um espaço no final de seus prompts (depois dos dois-pontos no exemplo anterior) para separar o prompt da resposta do usuário e deixar claro em que lugar o usuário deve fornecer seu texto. Por exemplo:

```
Please enter your name: Eric
Hello, Eric!
```

Às vezes, você vai querer escrever um prompt que seja maior que uma linha. Por exemplo, talvez você queira explicar ao usuário por que está pedindo determinada entrada. Você pode armazenar seu prompt em uma variável e passá-la para a função `input()`. Isso permite criar seu prompt com várias linhas e escrever uma instrução `input()` clara.

greeter.py

```
prompt = "If you tell us who you are, we can personalize the messages
you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print("\nHello, " + name + "!")
```

Esse exemplo mostra uma maneira de criar uma string multilinha. A primeira linha armazena a parte inicial da mensagem na variável `prompt`. Na segunda linha, o operador `+=` acrescenta a nova string no final da string que estava armazenada em `prompt`.

O prompt agora ocupa duas linhas, novamente, com um espaço após o ponto de interrogação por questões de clareza:

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Usando int() para aceitar entradas numéricas

Se usarmos a função `input()`, Python interpretará tudo que o usuário fornecer como uma string. Considere a sessão de interpretador a seguir, que pergunta a idade do usuário:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

O usuário digita o número 21, mas quando pedimos o valor de `age`, Python devolve `'21'`, que é a representação em string do valor numérico fornecido. Sabemos que Python interpretou a entrada como uma string porque o número agora está entre aspas. Se tudo que você quiser fazer é exibir a entrada, isso funcionará bem. Entretanto, se tentar usar a entrada como um número, você obterá um erro:

```
  >>> age = input("How old are you? ")
  How old are you? 21
u >>> age >= 18
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
v TypeError: unorderable types: str() >= int()
```

Quando tentamos usar a entrada para fazer uma comparação numérica u, Python gera um erro, pois não é capaz de comparar uma string com um inteiro: a string `'21'` armazenada em `age` não pode ser comparada ao valor numérico `18` v.

Podemos resolver esse problema usando a função `int()`, que diz a Python para tratar a entrada como um valor numérico. A função `int()` converte a representação em string de um número em uma representação numérica, como vemos a seguir:

```
  >>> age = input("How old are you? ")
  How old are you? 21
u >>> age = int(age)
  >>> age >= 18
  True
```

Nesse exemplo, quando fornecemos `21` no prompt, Python interpreta o número como uma string, mas o valor é então convertido para uma representação numérica por `int()` u. Agora Python pode executar o teste condicional: comparar `age` (que contém o valor numérico 21) com `18` para ver se `age` é maior ou igual a 18. Esse teste é avaliado como `True`.

Como podemos usar a função `int()` em um programa de verdade? Considere um programa que determina se as pessoas têm altura suficiente para andar em uma montanha-russa:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 36:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

O programa é capaz de comparar `height` com `36` porque `height = int(height)` converte o valor de entrada em uma representação numérica antes de fazer a comparação. Se o número fornecido for maior ou igual a 36, diremos ao usuário que sua altura é suficiente:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

Quando usar uma entrada numérica para fazer cálculos e comparações, lembre-se de converter o valor da entrada em uma representação numérica antes.

Operador de módulo

Uma ferramenta útil para trabalhar com informações numéricas é o *operador de módulo* (`%`), que divide um número por outro e devolve o resto:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

O operador de módulo não diz quantas vezes um número cabe em outro; ele simplesmente informa o resto.

Quando um número é divisível por outro, o resto é 0, portanto o operador de módulo sempre devolve 0 nesse caso. Podemos usar esse fato para determinar se um número é par ou ímpar:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd:
")
number = int(number)

if number % 2 == 0:
    print("\nThe number " + str(number) + " is even.")
else:
    print("\nThe number " + str(number) + " is odd.")
```

Números pares são sempre divisíveis por dois, portanto, se o módulo entre um número e dois for zero (nesse caso, `if number % 2 == 0`), o número será par. Caso contrário, será ímpar.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

Aceitando entradas em Python 2.7

Se você usa Python 2.7, utilize a função `raw_input()` quando pedir uma entrada ao usuário. Essa função interpreta todas as entradas como uma string, como faz `input()` em Python 3.

Python 2.7 também tem uma função `input()`, mas essa função interpreta a entrada do usuário como código Python e tenta executá-la. No melhor caso, você verá um erro informando que Python não é capaz de compreender a entrada; no pior caso, executará um código que não pretendia executar. Se estiver usando Python 2.7, utilize `raw_input()` no lugar de `input()`.

FAÇA VOCÊ MESMO

7.1 – Locação de automóveis: Escreva um programa que pergunte ao usuário qual tipo de carro ele gostaria de alugar. Mostre uma mensagem sobre esse carro, por exemplo, "Deixe-me ver se consigo um Subaru para você."

7.2 – Lugares em um restaurante: Escreva um programa que pergunte ao usuário quantas pessoas estão em seu grupo para jantar. Se a resposta for maior que oito, exiba uma mensagem dizendo que eles deverão esperar uma mesa. Caso contrário, informe que sua mesa está pronta.

7.3 – Múltiplos de dez: Peça um número ao usuário e, em seguida, informe se o número é múltiplo de dez ou não.

Introdução aos laços while

O laço `for` toma uma coleção de itens e executa um bloco de código uma vez para cada item da coleção. Em comparação, o laço `while` executa durante o tempo em que, ou *enquanto*, uma determinada condição for verdadeira.

Laço while em ação

Podemos usar um laço `while` para contar uma série de números. Por exemplo, o laço `while` a seguir conta de 1 a 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

Na primeira linha começamos a contar de 1 ao definir o valor de `current_number` com 1. O laço `while` é então configurado para continuar executando enquanto o valor de `current_number` for menor ou igual a 5. O código no laço exibe o valor `current_number` e então soma 1 a esse valor com `current_number += 1`. (O operador `+=` é um atalho para `current_number = current_number + 1`.)

Python repete o laço enquanto a condição `current_number <= 5` for verdadeira. Como 1 é menor que 5, Python exibe `1` e então soma 1, fazendo o número atual ser igual a `2`. Como 2 é menor que 5, Python exibe `2` e soma 1 novamente, fazendo o número atual ser igual a `3`, e assim sucessivamente.

Quando o valor de `current_number` for maior que 5, o laço para de executar e o programa termina:

```
1
2
3
4
5
```

Os programas que você usa no dia a dia provavelmente contêm laços `while`. Por exemplo, um jogo precisa de um laço `while` para continuar executando enquanto você quiser jogar, e pode parar de executar assim que você pedir para sair. Os programas não seriam divertidos se parassem de executar antes que lhes disséssemos para parar ou se continuassem executando mesmo depois que quiséssemos sair, portanto os laços `while` são bem úteis.

Deixando o usuário decidir quando quer sair

Podemos fazer o programa *parrot.py* executar enquanto o usuário quiser colocar a maior parte dele em um laço `while`. Definiremos um *valor de saída* e então deixaremos o programa executando enquanto o usuário não tiver fornecido o valor de saída:

parrot.py

```
u prompt = "\nTell me something, and I will repeat it back to you:"
  prompt += "\nEnter 'quit' to end the program. "

v message = ""
w while message != 'quit':
      message = input(prompt)
      print(message)
```

Em u definimos um prompt que informa quais são as duas opções ao usuário: fornecer uma mensagem ou o valor de saída (nesse caso, é `'quit'`). Em seguida, preparamos uma variável `message` v para armazenar o valor que o usuário fornecer. Definimos `message` como uma string vazia, `""`, de modo que Python tenha algo para conferir na primeira vez que alcançar a linha

com while. Na primeira vez que o programa executar e Python alcançar a instrução while, ele deverá comparar o valor de message com 'quit', mas o usuário ainda não forneceu nenhuma entrada. Se Python não tiver nada para comparar, ele não será capaz de continuar executando o programa. Para resolver esse problema, garantimos que message receba um valor inicial. Embora seja apenas uma string vazia, ela fará sentido para Python e permitirá que a comparação que faz o laço while funcionar seja feita. Esse laço while W executa enquanto o valor de message não for 'quit'.

Na primeira passagem pelo laço, message é apenas uma string vazia, portanto Python entra no laço. Em message = input(prompt), Python exibe o prompt e espera o usuário fornecer uma entrada. O que quer que seja fornecido será armazenado em message e exibido; em seguida, Python avalia novamente a condição na instrução while. Desde que o usuário não tenha fornecido a palavra 'quit', o prompt será exibido novamente e Python esperará mais entradas. Quando o usuário finalmente digitar 'quit', Python para de executar o laço while e o programa termina:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

Esse programa funciona bem, exceto que exibe a palavra 'quit' como se fosse uma mensagem de verdade. Um teste if simples corrige esse problema:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
```

```
message = input(prompt)

if message != 'quit':
    print(message)
```

Agora o programa faz uma verificação rápida antes de mostrar a mensagem e só a exibirá se ela não for igual ao valor de saída:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Usando uma flag

No exemplo anterior, tínhamos um programa que executava determinadas tarefas enquanto uma dada condição era verdadeira. E como ficaria em programas mais complicados, em que muitos eventos diferentes poderiam fazer o programa parar de executar?

Por exemplo, em um jogo, vários eventos diferentes podem encerrá-lo. Quando o jogador ficar sem espaçonaves, seu tempo se esgotar ou as cidades que ele deveria proteger forem todas destruídas, o jogo deverá terminar. O jogo termina se qualquer um desses eventos ocorrer. Se muitos eventos possíveis puderem ocorrer para o programa terminar, tentar testar todas essas condições em uma única instrução `while` torna-se complicado e difícil.

Para um programa que deva executar somente enquanto muitas condições forem verdadeiras, podemos definir uma variável que determina se o programa como um todo deve estar ativo. Essa variável, chamada de *flag*, atua como um sinal para o programa. Podemos escrever nossos programas de modo que executem enquanto a flag estiver definida com `True` e parem de executar quando qualquer um dos vários eventos definir o

valor da flag com `False`. Como resultado, nossa instrução `while` geral precisa verificar apenas uma condição: se a flag, no momento, é `True`. Então todos os nossos demais testes (para ver se um evento que deve definir a flag com `False` ocorreu) podem estar bem organizados no restante do programa.

Vamos adicionar uma flag em *parrot.py*, que vimos na seção anterior. Essa flag, que chamaremos de `active` (embora você possa lhe dar o nome que quiser), controlará se o programa deve ou não continuar executando:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

u active = True
v while active:
    message = input(prompt)

w     if message == 'quit':
        active = False
x     else:
        print(message)
```

Definimos a variável `active` com `True` **u** para que o programa comece em um estado ativo. Fazer isso simplifica a instrução `while`, pois nenhuma comparação é feita nessa instrução; a lógica é tratada em outras partes do programa. Enquanto a variável `active` permanecer `True`, o laço continuará a executar **v**.

Na instrução `if` contida no laço `while`, verificamos o valor de `message` depois que o usuário fornece sua entrada. Se o usuário fornecer `'quit'` **w**, definimos `active` com `False` e o laço `while` é encerrado. Se o usuário fornecer outro dado que não seja `'quit'` **x**, exibimos essa entrada como uma mensagem.

Esse programa gera a mesma saída do exemplo anterior, em que havíamos colocado o teste condicional diretamente na instrução `while`. Porém, agora que temos uma flag para indicar se o programa como um todo está em um estado ativo, será mais fácil acrescentar outros testes (por exemplo, instruções `elif`) para eventos que devam fazer `active` se tornar `False`. Isso é útil em programas complicados, como jogos, em que pode

haver muitos eventos, e qualquer um deles poderia fazer o programa parar de executar. Quando um desses eventos fizer a flag `active` se tornar `False`, o laço principal do jogo terminará, uma mensagem de *Game Over* poderia ser exibida e o jogador poderia ter a opção de jogar novamente.

Usando break para sair de um laço

Para sair de um laço `while` de imediato, sem executar qualquer código restante no laço, independentemente do resultado de qualquer teste condicional, utilize a instrução `break`. A instrução `break` direciona o fluxo de seu programa; podemos usá-la para controlar quais linhas de código são ou não são executadas, de modo que o programa execute apenas o código que você quiser, quando você quiser.

Por exemplo, considere um programa que pergunte aos usuários os nomes de lugares que eles já visitaram. Podemos interromper o laço `while` nesse programa chamando `break` assim que o usuário fornecer o valor `'quit'`:

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

u while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I'd love to go to " + city.title() + "!")
```

Um laço que comece com `while True` **u** executará indefinidamente, a menos que alcance uma instrução `break`. O laço desse programa continuará pedindo aos usuários para que entrem com os nomes das cidades em que eles estiveram até que `'quit'` seja fornecido. Quando `'quit'` for digitado, a instrução `break` é executada, fazendo Python sair do laço:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
```

```
I'd love to go to New York!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

NOTA Você pode usar a instrução `break` em qualquer laço de Python. Por exemplo, `break` pode ser usado para sair de um laço `for` que esteja percorrendo uma lista ou um dicionário.

Usando continue em um laço

Em vez de sair totalmente de um laço sem executar o restante de seu código, podemos usar a instrução `continue` para retornar ao início, com base no resultado de um teste condicional. Por exemplo, considere um laço que conte de 1 a 10, mas apresente apenas os números ímpares desse intervalo:

counting.py

```
  current_number = 0
  while current_number < 10:
u     current_number += 1
      if current_number % 2 == 0:
          continue

      print(current_number)
```

Inicialmente, definimos `current_number` com 0. Como esse valor é menor que 10, Python entra no laço `while`. Uma vez no laço, incrementamos o contador de 1 em **u**, portanto `current_number` passa a ser 1. A instrução `if` então verifica o módulo entre `current_number` e 2. Se o módulo for 0 (o que significa que `current_number` é divisível por 2), a instrução `continue` diz a Python para ignorar o restante do laço e voltar ao início. Se o número atual não for divisível por 2, o restante do laço será executado e Python exibirá o número atual:

```
1
```

```
3
5
7
9
```

Evitando loops infinitos

Todo laço `while` precisa de uma maneira de interromper a execução para que não continue executando indefinidamente. Por exemplo, o laço a seguir deve fazer a contagem de 1 a 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Contudo, se você omitir a linha `x += 1` (como vemos a seguir) por acidente, o laço executará para sempre:

```
# Esse laço executa indefinidamente!
x = 1
while x <= 5:
    print(x)
```

Agora o valor de `x` começará em `1`, mas jamais será modificado. Como resultado, o teste condicional `x <= 5` será sempre avaliado como `True` e o laço `while` executará indefinidamente, exibindo uma série de 1s, assim:

```
1
1
1
1
--trecho omitido--
```

Todo programador escreve ocasionalmente um loop infinito (ou laço infinito) com `while` por acidente, em especial quando os laços do programa tiverem condições de saída sutis. Se seu programa ficar preso em um loop infinito, tecle CTRL-C ou simplesmente feche a janela do terminal que está exibindo a saída de seu programa.

Para evitar escrever loops infinitos, teste todos os laços `while` e certifique-se de que eles serão encerrados conforme esperado. Se quiser que seu programa termine quando o usuário fornecer determinado valor de entrada, execute o programa e forneça esse valor. Se o programa não terminar, analise cuidadosamente o modo como seu programa trata o valor que deveria fazer o laço parar. Garanta que pelo menos uma parte do programa possa fazer a condição do laço ser `False` ou fazer uma instrução `break` ser alcançada.

NOTA Alguns editores, como o Sublime Text, tem uma janela de saída incluída. Isso pode dificultar a interrupção de um loop infinito, e talvez seja necessário fechar o editor para encerrar o laço.

FAÇA VOCÊ MESMO

7.4 – Ingredientes para uma pizza: Escreva um laço que peça ao usuário para fornecer uma série de ingredientes para uma pizza até que o valor `'quit'` seja fornecido. À medida que cada ingrediente é especificado, apresente uma mensagem informando que você acrescentará esse ingrediente à pizza.

7.5 – Ingressos para o cinema: Um cinema cobra preços diferentes para os ingressos de acordo com a idade de uma pessoa. Se uma pessoa tiver menos de 3 anos de idade, o ingresso será gratuito; se tiver entre 3 e 12 anos, o ingresso custará 10 dólares; se tiver mais de 12 anos, o ingresso custará 15 dólares. Escreva um laço em que você pergunte a idade aos usuários e, então, informe-lhes o preço do ingresso do cinema.

7.6 – Três saídas: Escreva versões diferentes do Exercício 7.4 ou do Exercício 7.5 que faça o seguinte, pelo menos uma vez:

- use um teste condicional na instrução `while` para encerrar o laço;
- use uma variável `active` para controlar o tempo que o laço executará;
- use uma instrução `break` para sair do laço quando o usuário fornecer o valor `'quit'`.

7.7 – Infinito: Escreva um laço que nunca termine e execute-o. (Para encerrar o laço, pressione CTRL-C ou feche a janela que está exibindo a saída.)

Usando um laço while com listas e dicionários

Até agora trabalhamos com apenas uma informação do usuário a cada vez. Recebemos a entrada do usuário e então a exibimos ou apresentamos uma

resposta a ela. Na próxima passagem pelo laço `while`, recebíamos outro valor de entrada e respondíamos a ela. Contudo, para controlar muitos usuários e informações, precisaremos usar listas e dicionários com nossos laços `while`.

Um laço `for` é eficiente para percorrer uma lista, mas você não deve modificar uma lista em um laço `for`, pois Python terá problemas para manter o controle dos itens da lista. Para modificar uma lista enquanto trabalhar com ela, utilize um laço `while`. Usar laços `while` com listas e dicionários permite coletar, armazenar e organizar muitas entradas a fim de analisá-las e apresentá-las posteriormente.

Transferindo itens de uma lista para outra

Considere uma lista de usuários recém-registrados em um site, porém não verificados. Depois de conferir esses usuários, como podemos transferi-los para uma lista separada de usuários confirmados? Uma maneira seria usar um laço `while` para extrair os usuários da lista de usuários não confirmados à medida que os verificarmos e então adicioná-los em uma lista separada de usuários confirmados. Esse código pode ter o seguinte aspecto:

confirmed_users.py

```
  # Começa com os usuários que precisam ser verificados,
  #  e com uma lista vazia para armazenar os usuários confirmados
u unconfirmed_users = ['alice', 'brian', 'candace']
  confirmed_users = []

  # Verifica cada usuário até que não haja mais usuários não confirmados
  #  Transfere cada usuário verificado para a lista de usuários
  confirmados
v while unconfirmed_users:
w     current_user = unconfirmed_users.pop()

      print("Verifying user: " + current_user.title())
x     confirmed_users.append(current_user)

  # Exibe todos os usuários confirmados
  print("\nThe following users have been confirmed:")
  for confirmed_user in confirmed_users:
```

```
    print(confirmed_user.title())
```

Começamos com uma lista de usuários não confirmados em u (Alice, Brian e Candace) e uma lista vazia para armazenar usuários confirmados. O laço `while` em v executa enquanto a lista `unconfirmed_users` não estiver vazia. Nesse laço, a função `pop()` em w remove os usuários não verificados, um de cada vez, do final de `unconfirmed_users`. Nesse caso, como Candace é o último elemento da lista `unconfirmed_users`, seu nome será o primeiro a ser removido, armazenado em `current_user` e adicionado à lista `confirmed_users` em x. O próximo é Brian e, depois, Alice.

Simulamos a confirmação de cada usuário exibindo uma mensagem de verificação e então adicionando-os à lista de usuários confirmados. À medida que a lista de usuários não confirmados diminui, a lista de usuários confirmados aumenta. Quando a lista de usuários não confirmados estiver vazia, o laço para e a lista de usuários confirmados é exibida:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Removendo todas as instâncias de valores específicos de uma lista

No Capítulo 3 usamos `remove()` para remover um valor específico de uma lista. A função `remove()` era apropriada porque o valor em que estávamos interessados aparecia apenas uma vez na lista. Porém, e se quiséssemos remover da lista todas as instâncias de um valor?

Suponha que tenhamos uma lista de animais de estimação com o valor `'cat'` repetido várias vezes. Para remover todas as instâncias desse valor, podemos executar um laço `while` até `'cat'` não estar mais na lista, como vemos aqui:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

Começamos com uma lista contendo várias instâncias de `'cat'`. Após exibir a lista, Python entra no laço `while`, pois encontra o valor `'cat'` na lista pelo menos uma vez. Depois que entrar no laço, Python remove a primeira instância de `'cat'`, retorna à linha `while` e então entra novamente no laço quando descobre que `'cat'` ainda está na lista. Cada instância de `'cat'` é removida até que esse valor não esteja mais na lista; nesse momento, Python sai do laço e exibe a lista novamente:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Preenchendo um dicionário com dados de entrada do usuário

Podemos pedir a quantidade de entrada que for necessária a cada passagem por um laço `while`. Vamos criar um programa de enquete em que cada passagem pelo laço solicita o nome do participante e uma resposta. Armazenaremos os dados coletados em um dicionário, pois queremos associar cada resposta a um usuário em particular:

mountain_poll.py

```
responses = {}

# Define uma flag para indicar que a enquete está ativa
polling_active = True

while polling_active:
    # Pede o nome da pessoa e a resposta
u     name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday?
")
```

```
    # Armazena a resposta no dicionário
v     responses[name] = response

      # Descobre se outra pessoa vai responder à enquete
w      repeat = input("Would you like to let another person respond? (yes/
no) ")
      if repeat == 'no':
          polling_active = False

  # A enquete foi concluída. Mostra os resultados
  print("\n--- Poll Results ---")
x for name, response in responses.items():
      print(name + " would like to climb " + response + ".")
```

O programa inicialmente define um dicionário vazio (`responses`) e cria uma flag (`polling_active`) para indicar que a enquete está ativa. Enquanto `polling_active` for `True`, Python executará o código que está no laço `while`.

Nesse laço é solicitado ao usuário que entre com seu nome e uma montanha que gostaria de escalar **u**. Essa informação é armazenada no dicionário `responses` **v**, e uma pergunta é feita ao usuário para saber se ele quer que a enquete continue **w**. Se o usuário responder `yes`, o programa entrará no laço `while` novamente. Se responder `no`, a flag `polling_active` será definida com `False`, o laço `while` para de executar e o último bloco de código em **x** exibe o resultado da enquete.

Se executar esse programa e fornecer exemplos de respostas, você deverá ver uma saída como esta:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

FAÇA VOCÊ MESMO

7.8 – Lanchonete: Crie uma lista chamada `sandwich_orders` e a preencha com os nomes de vários sanduíches. Em seguida, crie uma lista vazia chamada `finished_sandwiches`. Percorra a lista de pedidos de sanduíches com um laço e mostre uma mensagem para cada pedido, por exemplo, `Preparei seu sanduíche de atum`. À medida que cada sanduíche for preparado, transfira-o para a lista de sanduíches prontos. Depois que todos os sanduíches estiverem prontos, mostre uma mensagem que liste cada sanduíche preparado.

7.9 – Sem pastrami: Usando a lista `sandwich_orders` do Exercício 7.8, garanta que o sanduíche de `'pastrami'` apareça na lista pelo menos três vezes. Acrescente um código próximo ao início de seu programa para exibir uma mensagem informando que a lanchonete está sem pastrami e, então, use um laço `while` para remover todas as ocorrências de `'pastrami'` e `sandwich_orders`. Garanta que nenhum sanduíche de pastrami acabe em `finished_sandwiches`.

7.10 – Férias dos sonhos: Escreva um programa que faça uma enquete sobre as férias dos sonhos dos usuários. Escreva um prompt semelhante a este: *Se pudesse visitar um lugar do mundo, para onde você iria?* Inclua um bloco de código que apresente os resultados da enquete.

Resumo

Neste capítulo aprendemos a usar `input()` para permitir que os usuários forneçam suas próprias informações em seus programas. Vimos como trabalhar com entradas tanto textuais quanto numéricas e a usar laços `while` para deixar seus programas executarem pelo tempo que os usuários quiserem. Conhecemos várias maneiras de controlar o fluxo de um laço `while`: definindo uma flag `active`, usando a instrução `break` e com a instrução `continue`. Aprendemos a usar um laço `while` para transferir itens de uma lista para outra e vimos como remover todas as instâncias de um valor de uma lista. Também vimos como laços `while` podem ser usados com dicionários.

No Capítulo 8, conheceremos as *funções*. As funções permitem dividir seus programas em partes menores, cada uma fazendo uma tarefa específica. Podemos chamar uma função quantas vezes quisermos e armazená-las em arquivos separados. Ao usar funções, podemos escrever

códigos mais eficientes, em que seja mais fácil resolver problemas e dar manutenção, com possibilidade de serem reutilizados em vários programas diferentes.

8
FUNÇÕES

Neste capítulo aprenderemos a escrever *funções*, que são blocos de código nomeados, concebidos para realizar uma tarefa específica. Quando queremos executar uma tarefa em particular, definida em uma função, *chamamos* o nome da função responsável por ela. Se precisar executar essa tarefa várias vezes durante seu programa, não será necessário digitar todo o código para a mesma tarefa repetidamente: basta chamar a função dedicada ao tratamento dessa tarefa e a chamada dirá a Python para executar o código da função. Você perceberá que usar funções permite escrever, ler, testar e corrigir seus programas de modo mais fácil.

Neste capítulo também veremos maneiras de passar informações às funções. Aprenderemos a escrever determinadas funções cuja tarefa principal seja exibir informações e outras funções que visam a processar dados e devolver um valor ou um conjunto de valores. Por fim, veremos como armazenar funções em arquivos separados, chamados de *módulos*, para ajudar a organizar os arquivos principais de seu programa.

Definindo uma função

Eis uma função simples chamada `greet_user()` que exibe uma saudação:

greeter.py

```
u def greet_user():
v     """Exibe uma saudação simples."""
w     print("Hello!")

x greet_user()
```

Esse exemplo mostra a estrutura mais simples possível para uma função.

A linha em u utiliza a palavra reservada `def` para informar Python que estamos definindo uma função. Essa é a *definição da função*, que informa o nome da função a Python e, se for aplicável, quais são os tipos de informação necessários à função para que ela faça sua tarefa. Os parênteses contêm essa informação. Nesse caso, o nome da função é `greet_user()`, e ela não precisa de nenhuma informação para executar sua tarefa, portanto os parênteses estão vazios. (Mesmo assim, eles são obrigatórios.) Por fim, a definição termina com dois-pontos.

Qualquer linha indentada após `def greet_user():` faz parte do *corpo* da função. O texto em v é um comentário chamado *docstring*, que descreve o que a função faz. As docstrings são colocadas entre aspas triplas, que Python procura quando gera a documentação das funções de seus programas.

A linha `print("Hello!")` w é a única linha com código propriamente dito no corpo dessa função, portanto `greet_user()` realiza apenas uma tarefa: `print("Hello!")`.

Quando quiser usar essa função, você deve chamá-la. Uma *chamada de função* diz a Python para executar o código da função. Para *chamar* uma função, escreva o nome dela, seguido de qualquer informação necessária entre parênteses, como vemos em x. Como nenhuma informação é necessária nesse caso, chamar nossa função é simples e basta fornecer `greet_user()`. Como esperado, ela exibe `Hello!`:

```
Hello!
```

Passando informações para uma função

Se for um pouco modificada, a função `greet_user()` não só pode dizer `Hello!` ao usuário como também pode saudá-lo pelo nome. Para que a função faça isso, especifique `username` entre os parênteses da definição da função em `def greet_user()`. Ao acrescentar `username` aqui, permitimos que a função aceite qualquer valor que você especificar para `username`. A função agora espera que um valor seja fornecido para `username` sempre que ela for chamada. Ao chamar `greet_user()`, você poderá lhe passar um nome, por exemplo, `'jesse'`, entre parênteses:

```
def greet_user(username):
    """Exibe uma saudação simples."""
    print("Hello, " + username.title() + "!")

greet_user('jesse')
```

Usar `greet_user('jesse')` faz `greet_user()` ser chamada e fornece as informações de que a função precisa para executar a instrução `print`. A função aceita o nome que você passar e exibe a saudação para esse nome:

```
Hello, Jesse!
```

De modo semelhante, usar `greet_user('sarah')` chama `greet_user()`, passa `'sarah'` a essa função e exibe `Hello, Sarah!`. Você pode chamar `greet_user()` quantas vezes quiser e lhe passar qualquer nome desejado de modo a gerar sempre uma saída previsível.

Argumentos e parâmetros

Na função `greet_user()` anterior, definimos `greet_user()` para que exija um valor para a variável `username`. Depois que chamamos a função e lhe fornecemos a informação (o nome de uma pessoa), a saudação correta foi exibida.

A variável `username` na definição de `greet_user()` é um exemplo de *parâmetro* – uma informação de que a função precisa para executar sua tarefa. O valor `'jesse'` em `greet_user('jesse')` é um exemplo de *argumento*. Um argumento é uma informação passada para uma função em sua chamada. Quando chamamos a função, colocamos entre parênteses o valor com que queremos que a função trabalhe. Nesse caso, o argumento `'jesse'` foi passado para a função `greet_user()` e o valor foi armazenado no parâmetro `username`.

NOTA Às vezes, as pessoas falam de argumentos e parâmetros de modo indistinto. Não fique surpreso se vir as variáveis de uma definição de função serem referenciadas como argumentos, ou as variáveis de uma chamada de função serem chamadas de parâmetros.

FAÇA VOCÊ MESMO

8.1 – Mensagem: Escreva uma função chamada `display_message()` que mostre uma frase informando a todos o que você está aprendendo neste capítulo. Chame a função e certifique-se de que a mensagem seja exibida corretamente.

8.2 – Livro favorito: Escreva uma função chamada `favorite_book()` que aceite um parâmetro `title`. A função deve exibir uma mensagem como `Um dos meus livros favoritos é Alice no país das maravilhas.` Chame a função e não se esqueça de incluir o título do livro como argumento na chamada da função.

Passando argumentos

Pelo fato de ser possível que uma definição de função tenha vários parâmetros, uma chamada de função pode precisar de diversos argumentos. Os argumentos podem ser passados para as funções de várias maneiras. Podemos usar *argumentos posicionais*, que devem estar na mesma ordem em que os parâmetros foram escritos, *argumentos nomeados* (keyword arguments), em que cada argumento é constituído de um nome de variável e de um valor, ou por meio de listas e dicionários de valores. Vamos analisar cada um deles.

Argumentos posicionais

Quando chamamos uma função, Python precisa fazer a correspondência entre cada argumento da chamada da função e um parâmetro da definição. A maneira mais simples de fazer isso é contar com a ordem dos argumentos fornecidos. Valores cuja correspondência seja feita dessa maneira são chamados de *argumentos posicionais*.

Para ver como isso funciona considere uma função que apresente informações sobre animais de estimação. A função nos informa o tipo de cada animal de estimação e o nome dele, como vemos aqui:

pets.py

```
u def describe_pet(animal_type, pet_name):
      """Exibe informações sobre um animal de estimação."""
      print("\nI have a " + animal_type + ".")
      print("My " + animal_type + "'s name is " + pet_name.title() + ".")

v describe_pet('hamster', 'harry')
```

A definição mostra que essa função precisa de um tipo de animal e de seu nome **u**. Quando chamamos `describe_pet()`, devemos fornecer o tipo do animal e um nome, nessa ordem. Por exemplo, na chamada da função, o argumento `'hamster'` é armazenado no parâmetro `animal_type` e o argumento `'harry'` é armazenado no parâmetro `pet_name` **v**. No corpo da função, esses dois parâmetros são usados para exibir informações sobre o animal de estimação descrito.

A saída apresenta um hamster chamado Harry:

```
I have a hamster.
My hamster's name is Harry.
```

Várias chamadas de função

Podemos chamar uma função quantas vezes forem necessárias. Descrever um segundo animal de estimação diferente exige apenas mais uma chamada a `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Nessa segunda chamada da função, passamos os argumentos `'dog'` e `'willie'` a `describe_pet()`. Assim como no conjunto anterior de argumentos que usamos, Python faz a correspondência entre `'dog'` e o parâmetro `animal_type` e entre `'willie'` e o parâmetro `pet_name`. Como antes, a função faz sua tarefa, porém, dessa vez, exibe valores para um cachorro chamado Willie. Agora temos um hamster chamado Harry e um cachorro chamado Willie:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Chamar uma função várias vezes é uma maneira eficiente de trabalhar. O código que descreve um animal de estimação é escrito uma só vez na função. Então, sempre que quiser descrever um novo animal de estimação, podemos chamar a função com as informações sobre esse animal. Mesmo que o código para descrever um animal de estimação fosse expandido atingindo dez linhas, poderíamos ainda descrever um novo animal de estimação chamando a função novamente com apenas uma linha.

Podemos usar tantos argumentos posicionais quantos forem necessários nas funções. Python trabalha com os argumentos fornecidos na chamada da função e faz a correspondência de cada um com o parâmetro associado na definição da função.

A ordem é importante em argumentos posicionais

Podemos obter resultados inesperados se confundirmos a ordem dos argumentos em uma chamada de função quando argumentos posicionais forem usados:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('harry', 'hamster')
```

Nessa chamada de função, listamos primeiro o nome e depois o tipo do animal. Como dessa vez o argumento `'harry'` foi definido antes, esse valor é armazenado no parâmetro `animal_type`. De modo semelhante, `'hamster'` é armazenado em `pet_name`. Agora temos um “harry” chamado “Hamster”:

```
I have a harry.
My harry's name is Hamster.
```

Se obtiver resultados engraçados como esse, verifique se a ordem dos argumentos em sua chamada de função corresponde à ordem dos parâmetros na definição da função.

Argumentos nomeados

Um *argumento nomeado* (keyword argument) é um par nome-valor passado para uma função. Associamos diretamente o nome e o valor no próprio argumento para que não haja confusão quando ele for passado para a função (você não acabará com um harry chamado Hamster). Argumentos nomeados fazem com que você não precise se preocupar com a ordem correta de seus argumentos na chamada da função e deixam claro o papel de cada valor na chamada.

Vamos reescrever *pets.py* usando argumentos nomeados para chamar `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(animal_type='hamster', pet_name='harry')
```

A função `describe_pet()` não mudou. Entretanto, quando chamamos a função, dizemos explicitamente a Python a qual parâmetro cada argumento deve corresponder. Quando Python lê a chamada da função, ele sabe que deve armazenar o argumento `'hamster'` no parâmetro `animal_type` e o argumento `'harry'` em `pet_name`. A saída mostra corretamente que temos um hamster chamado Harry.

A ordem dos argumentos nomeados não importa, pois Python sabe o que é cada valor. As duas chamadas de função a seguir são equivalentes:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTA Quando usar argumentos nomeados, lembre-se de usar os nomes exatos dos parâmetros usados na definição da função.

Valores default

Ao escrever uma função, podemos definir um *valor default* para cada parâmetro. Se um argumento para um parâmetro for especificado na chamada da função, Python usará o valor desse argumento. Se não for, o valor default do parâmetro será utilizado. Portanto, se um valor default for

definido para um parâmetro, você poderá excluir o argumento correspondente, que normalmente seria especificado na chamada da função. Usar valores default pode simplificar suas chamadas de função e deixar mais claro o modo como suas funções normalmente são utilizadas.

Por exemplo, se perceber que a maioria das chamadas a `describe_pet()` é usada para descrever cachorros, você pode definir o valor default de `animal_type` com `'dog'`. Agora qualquer pessoa que chamar `describe_pet()` para um cachorro poderá omitir essa informação:

```
def describe_pet(pet_name, animal_type='dog'):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(pet_name='willie')
```

Mudamos a definição de `describe_pet()` para incluir um valor default igual a `'dog'` para `animal_type`. A partir de agora, quando a função for chamada sem um `animal_type` especificado, Python saberá que deve usar o valor `'dog'` para esse parâmetro:

```
I have a dog.
My dog's name is Willie.
```

Observe que a ordem dos parâmetros na definição da função precisou ser alterada. Como o uso do valor default faz com que não seja necessário especificar um tipo de animal como argumento, o único argumento restante na chamada da função é o nome do animal de estimação. Python continua interpretando esse valor como um argumento posicional, portanto, se a função for chamada somente com o nome de um animal de estimação, esse argumento corresponderá ao primeiro parâmetro listado na definição da função. Esse é o motivo pelo qual o primeiro parâmetro deve ser `pet_name`.

O modo mais simples de usar essa função agora é fornecer apenas o nome de um cachorro na chamada da função:

```
describe_pet('willie')
```

Essa chamada de função produzirá a mesma saída do exemplo anterior. O único argumento fornecido é `'willie'`, portanto ele é associado ao primeiro parâmetro da definição da função, que é `pet_name`. Como nenhum argumento foi fornecido para `animal_type`, Python usa o valor default, que é `'dog'`.

Para descrever um animal que não seja um cachorro, uma chamada de função como esta pode ser usada:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Como um argumento explícito para `animal_type` foi especificado, Python ignorará o valor default do parâmetro.

NOTA Ao usar valores default, qualquer parâmetro com um valor desse tipo deverá ser listado após todos os parâmetros que não tenham valores default. Isso permite que Python continue a interpretar os argumentos posicionais corretamente.

Chamadas de função equivalentes

Como os argumentos posicionais, os argumentos nomeados e os valores default podem ser usados em conjunto, e com frequência você terá várias maneiras equivalentes de chamar uma função. Considere a definição a seguir de `describe_pets()` com um valor default especificado:

```
def describe_pet(pet_name, animal_type='dog'):
```

Com essa definição, um argumento sempre deverá ser fornecido para `pet_name` e esse valor pode ser especificado por meio do formato posicional ou nomeado. Se o animal descrito não for um cachorro, um argumento para `animal_type` deverá ser incluído na chamada, e esse argumento também pode ser especificado com o formato posicional ou nomeado.

Todas as chamadas a seguir serão adequadas a essa função:

```
# Um cachorro chamado Willie
describe_pet('willie')
describe_pet(pet_name='willie')

# Um hamster chamado Harry
describe_pet('harry', 'hamster')
```

```
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Cada uma dessas chamadas de função produzirá a mesma saída dos exemplos anteriores.

NOTA O estilo de chamada que você usar realmente não importa. Desde que suas chamadas de função gerem a saída desejada, basta usar o estilo que achar mais fácil de entender.

Evitando erros em argumentos

Quando começar a usar funções, não se surpreenda se você se deparar com erros sobre argumentos sem correspondência. Argumentos sem correspondência ocorrem quando fornecemos menos ou mais argumentos necessários à função para que ela realize sua tarefa. Por exemplo, eis o que acontece se tentarmos chamar `describe_pet()` sem argumentos:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet()
```

Python reconhece que algumas informações estão faltando na chamada da função e o traceback nos informa quais são:

```
  Traceback (most recent call last):
u   File "pets.py", line 6, in <module>
v     describe_pet()
w TypeError: describe_pet() missing 2 required positional arguments:
'animal_
  type' and 'pet_name'
```

Em u o traceback nos informa em que local está o problema, o que nos permite olhar o código novamente e ver que algo deu errado em nossa chamada de função. Em v a chamada de função causadora do problema é apresentada. Em w o traceback nos informa que há dois argumentos ausentes na chamada da função e mostra o nome desses argumentos. Se essa função estivesse em um arquivo separado, provavelmente poderíamos

reescrever a chamada de forma correta sem precisar abrir esse arquivo e ler o código da função.

Python nos ajuda lendo o código da função e informando os nomes dos argumentos que devemos fornecer. Esse é outro motivo para dar nomes descritivos às suas variáveis e funções. Se fizer isso, as mensagens de erro de Python serão mais úteis para você e para qualquer pessoa que possa usar o seu código.

Se você fornecer argumentos demais, deverá obter um traceback semelhante, que poderá ajudá-lo a fazer uma correspondência correta entre a chamada da função e sua definição.

FAÇA VOCÊ MESMO

8.3 – Camiseta: Escreva uma função chamada `make_shirt()` que aceite um tamanho e o texto de uma mensagem que deverá ser estampada na camiseta. A função deve exibir uma frase que mostre o tamanho da camiseta e a mensagem estampada.

Chame a função uma vez usando argumentos posicionais para criar uma camiseta. Chame a função uma segunda vez usando argumentos nomeados.

8.4 – Camisetas grandes: Modifique a função `make_shirt()` de modo que as camisetas sejam grandes por default, com uma mensagem *Eu amo Python*. Crie uma camiseta grande e outra média com a mensagem default, e uma camiseta de qualquer tamanho com uma mensagem diferente.

8.5 – Cidades: Escreva uma função chamada `describe_city()` que aceite o nome de uma cidade e seu país. A função deve exibir uma frase simples, como `Reykjavik está localizada na Islândia`. Forneça um valor default ao parâmetro que representa o país. Chame sua função para três cidades diferentes em que pelo menos uma delas não esteja no país default.

Valores de retorno

Uma função nem sempre precisa exibir sua saída diretamente. Em vez disso, ela pode processar alguns dados e então devolver um valor ou um conjunto de valores. O valor devolvido pela função é chamado de *valor de retorno*. A instrução `return` toma um valor que está em uma função e o envia de volta à linha que a chamou. Valores de retorno permitem passar boa parte do trabalho pesado de um programa para funções, o que pode

simplificar o corpo de seu programa.

Devolvendo um valor simples

Vamos observar uma função que aceite um primeiro nome e um sobrenome e devolva um nome completo formatado de modo elegante:

formatted_name.py

```
u def get_formatted_name(first_name, last_name):
      """Devolve um nome completo formatado de modo elegante."""
v     full_name = first_name + ' ' + last_name
w      return full_name.title()

x musician = get_formatted_name('jimi', 'hendrix')
  print(musician)
```

A definição de `get_formatted_name()` aceita um primeiro nome e um sobrenome u como parâmetros. A função combina esses dois nomes, acrescenta um espaço entre eles e armazena o resultado em `full_name` v. O valor de `full_name` é convertido para que tenha letras iniciais maiúsculas e é devolvido para a linha que fez a chamada em w.

Quando chamamos uma função que devolve um valor, precisamos fornecer uma variável em que o valor de retorno possa ser armazenado. Nesse caso, o valor devolvido é armazenado na variável `musician` em x. A saída mostra um nome formatado de modo elegante, composto das partes do nome de uma pessoa:

```
Jimi Hendrix
```

Pode parecer bastante trabalho para obter um nome formatado de forma elegante quando poderíamos simplesmente ter escrito:

```
print("Jimi Hendrix")
```

No entanto, quando consideramos trabalhar com um programa de grande porte, que precise armazenar muitos primeiros nomes e sobrenomes separadamente, funções como `get_formatted_name()` tornam-se muito convenientes. Armazenamos os primeiros nomes e os sobrenomes de forma separada e então chamamos essa função sempre que quisermos

exibir um nome completo.

Deixando um argumento opcional

Às vezes, faz sentido criar um argumento opcional para que as pessoas que usarem a função possam optar por fornecer informações extras somente se quiserem. Valores default podem ser usados para deixar um argumento opcional.

Por exemplo, suponha que queremos expandir `get_formatted_name()` para que trate nomes do meio também. Uma primeira tentativa para incluir nomes do meio poderia ter o seguinte aspecto:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Devolve um nome completo formatado de modo elegante."""
    full_name = first_name + ' ' + middle_name + ' ' + last_name
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Essa função é apropriada quando fornecemos um primeiro nome, um nome do meio e um sobrenome. Ela aceita todas as três partes de um nome e então compõe uma string a partir delas. A função acrescenta espaços nos pontos apropriados e converte o nome completo para que as iniciais sejam maiúsculas:

```
John Lee Hooker
```

No entanto, nomes do meio nem sempre são necessários, e essa função, conforme está escrita, não seria apropriada se tentássemos chamá-la somente com um primeiro nome e um sobrenome. Para deixar o nome do meio opcional, podemos associar um valor default vazio ao argumento `middle_name` e ignorá-lo, a menos que o usuário forneça um valor. Para que `get_formatted_name()` funcione sem um nome do meio, definimos o valor default de `middle_name` com uma string vazia e o passamos para o final da lista de parâmetros:

```
u def get_formatted_name(first_name, last_name, middle_name=''):
      """Devolve um nome completo formatado de modo elegante."""
```

```
v     if middle_name:
          full_name = first_name + ' ' + middle_name + ' ' + last_name
w     else:
          full_name = first_name + ' ' + last_name
      return full_name.title()

  musician = get_formatted_name('jimi', 'hendrix')
  print(musician)

x musician = get_formatted_name('john', 'hooker', 'lee')
  print(musician)
```

Nesse exemplo, o nome é criado a partir de três partes possíveis. Como o primeiro nome e o sobrenome sempre existem, esses parâmetros são listados antes na definição da função. O nome do meio é opcional, portanto é listado por último na definição, e seu valor default é uma string vazia `u`.

No corpo da função verificamos se um nome do meio foi especificado. Python interpreta strings não vazias como `True`, portanto `if middle_name` será avaliado como `True` se um argumento para o nome do meio estiver na chamada da função `v`. Se um nome do meio for especificado, o primeiro nome, o nome do meio e o sobrenome serão combinados para compor um nome completo. Esse nome é então alterado para que as iniciais sejam maiúsculas e é devolvido para a linha que chamou a função: ele será armazenado em uma variável `musician` e exibido. Se um nome do meio não for especificado, a string vazia falhará no teste `if` e o bloco `else` será executado `w`. O nome completo será composto apenas do primeiro nome e do sobrenome, e o nome formatado é devolvido para a linha que fez a chamada: ele será armazenado em `musician` e exibido.

Chamar essa função com um primeiro nome e um sobrenome é simples. Se usarmos um nome do meio, porém, precisamos garantir que esse nome seja o último argumento passado para que Python faça a correspondência dos argumentos posicionais de forma correta `x`.

Essa versão modificada de nossa função é apropriada para pessoas que tenham apenas um primeiro nome e um sobrenome, mas funciona também para pessoas que tenham um nome do meio:

```
Jimi Hendrix
John Lee Hooker
```

Valores opcionais permitem que as funções tratem uma grande variedade de casos de uso, ao mesmo tempo que simplificam ao máximo as chamadas de função.

Devolvendo um dicionário

Uma função pode devolver qualquer tipo de valor necessário, incluindo estruturas de dados mais complexas como listas e dicionários. Por exemplo, a função a seguir aceita partes de um nome e devolve um dicionário que representa uma pessoa:

person.py

```
  def build_person(first_name, last_name):
      """Devolve um dicionário com informações sobre uma pessoa."""
u     person = {'first': first_name, 'last': last_name}
v     return person

  musician = build_person('jimi', 'hendrix')
w print(musician)
```

A função `build_person()` aceita um primeiro nome e um sobrenome e então reúne esses valores em um dicionário em u. O valor de `first_name` é armazenado com a chave `'first'` e o valor de `last_name` é armazenado com a chave `'last'`. O dicionário completo que representa a pessoa é devolvido em v. O valor de retorno é exibido em w, com as duas informações textuais originais agora armazenadas em um dicionário:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Essa função aceita informações textuais simples e as coloca em uma estrutura de dados mais significativa, que permite trabalhar com as informações além de simplesmente exibi-las. As strings `'jimi'` e `'hendrix'` agora estão identificadas como um primeiro nome e um sobrenome. Podemos facilmente estender essa função para que aceite valores opcionais como um nome do meio, uma idade, uma profissão ou qualquer outra informação que você queira armazenar sobre uma pessoa. Por exemplo, a

alteração a seguir permite armazenar a idade de uma pessoa também:

```
def build_person(first_name, last_name, age=''):
    """Devolve um dicionário com informações sobre uma pessoa."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

Adicionamos um novo parâmetro opcional `age` à definição da função e atribuímos um valor default vazio ao parâmetro. Se a chamada da função incluir um valor para esse parâmetro, ele será armazenado no dicionário. Essa função sempre armazena o nome de uma pessoa, mas também pode ser modificada para guardar outras informações que você quiser sobre ela.

Usando uma função com um laço while

Podemos usar funções com todas as estruturas Python que conhecemos até agora. Por exemplo, vamos usar a função `get_formatted_name()` com um laço `while` para saudar os usuários de modo mais formal. Eis uma primeira tentativa de saudar pessoas usando seu primeiro nome e o sobrenome:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Devolve um nome completo formatado de modo elegante."""
    full_name = first_name + ' ' + last_name
    return full_name.title()

# Este é um loop infinito!
while True:
u     print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print("\nHello, " + formatted_name + "!")
```

Nesse exemplo, usamos uma versão simples de `get_formatted_name()` que não envolve nomes do meio. O laço `while` pede que o usuário forneça seu nome; além disso, solicitamos o primeiro nome e o sobrenome separadamente **u**.

Porém há um problema com esse laço `while`: não definimos uma condição de saída. Onde devemos colocar uma condição de saída quando pedimos uma série de entradas? Queremos que o usuário seja capaz de sair o mais facilmente possível, portanto cada prompt deve oferecer um modo de fazer isso. A instrução `break` permite um modo simples de sair do laço em qualquer prompt:

```
def get_formatted_name(first_name, last_name):
    """Devolve um nome completo formatado de modo elegante."""
    full_name = first_name + ' ' + last_name
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print("\nHello, " + formatted_name + "!")
```

Adicionamos uma mensagem que informa como o usuário pode sair e então encerramos o laço se o usuário fornecer o valor de saída em qualquer um dos prompts. Agora o programa continuará saudando as pessoas até que alguém forneça `'q'` em algum dos nomes:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
```

```
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

FAÇA VOCÊ MESMO

8.6 – Nomes de cidade: Escreva uma função chamada `city_country()` que aceite o nome de uma cidade e seu país. A função deve devolver uma string formatada assim:

```
"Santiago, Chile"
```

Chame sua função com pelo menos três pares cidade-país e apresente o valor devolvido.

8.7 – Álbum: Escreva uma função chamada `make_album()` que construa um dicionário descrevendo um álbum musical. A função deve aceitar o nome de um artista e o título de um álbum e deve devolver um dicionário contendo essas duas informações. Use a função para criar três dicionários que representem álbuns diferentes. Apresente cada valor devolvido para mostrar que os dicionários estão armazenando as informações do álbum corretamente.

Acrescente um parâmetro opcional em `make_album()` que permita armazenar o número de faixas em um álbum. Se a linha que fizer a chamada incluir um valor para o número de faixas, acrescente esse valor ao dicionário do álbum. Faça pelo menos uma nova chamada da função incluindo o número de faixas em um álbum.

8.8 – Álbuns dos usuários: Comece com o seu programa do Exercício 8.7. Escreva um laço `while` que permita aos usuários fornecer o nome de um artista e o título de um álbum. Depois que tiver essas informações, chame `make_album()` com as entradas do usuário e apresente o dicionário criado. Lembre-se de incluir um valor de saída no laço `while`.

Passando uma lista para uma função

Com frequência, você achará útil passar uma lista para uma função, seja uma lista de nomes, de números ou de objetos mais complexos, como dicionários. Se passarmos uma lista a uma função, ela terá acesso direto ao conteúdo dessa lista. Vamos usar funções para que o trabalho com listas

seja mais eficiente.

Suponha que tenhamos uma lista de usuários e queremos exibir uma saudação a cada um. O exemplo a seguir envia uma lista de nomes a uma função chamada `greet_users()`, que saúda cada pessoa da lista individualmente:

greet_users.py

```
  def greet_users(names):
      """Exibe uma saudação simples a cada usuário da lista."""
      for name in names:
          msg = "Hello, " + name.title() + "!"
          print(msg)

u usernames = ['hannah', 'ty', 'margot']
  greet_users(usernames)
```

Definimos `greet_users()` para que espere uma lista de nomes, que é armazenada no parâmetro `names`. A função percorre a lista recebida com um laço e exibe uma saudação para cada usuário. Em u definimos uma lista de usuários e então passamos a lista `usernames` para `greet_users()` em nossa chamada de função:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Essa é a saída que queríamos. Todo usuário vê uma saudação personalizada, e você pode chamar a função sempre que quiser para saudar um conjunto específico de usuários.

Modificando uma lista em uma função

Quando passamos uma lista a uma função, ela pode ser modificada. Qualquer alteração feita na lista no corpo da função é permanente, permitindo trabalhar de modo eficiente, mesmo quando lidamos com grandes quantidades de dados.

Considere uma empresa que cria modelos de designs submetidos pelos usuários e que são impressos em 3D. Os designs são armazenados em uma

lista e, depois de impressos, são transferidos para uma lista separada. O código a seguir faz isso sem usar funções:

printing_models.py

```
# Começa com alguns designs que devem ser impressos
unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simula a impressão de cada design, até que não haja mais nenhum
#  Transfere cada design para completed_models após a impressão
while unprinted_designs:
    current_design = unprinted_designs.pop()

    # Simula a criação de uma impressão 3D a partir do design
    print("Printing model: " + current_design)
    completed_models.append(current_design)

# Exibe todos os modelos finalizados
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

Esse programa começa com uma lista de designs que devem ser impressos e uma lista vazia chamada `completed_models` para a qual cada design será transferido após a impressão. Enquanto houver designs em `unprinted_designs`, o laço `while` simulará a impressão de cada um deles removendo um design do final da lista, armazenando-o em `current_design` e exibindo uma mensagem informando que o design atual está sendo impresso. O design então é adicionado à lista de modelos finalizados. Quando o laço acaba de executar, uma lista de designs impressos é exibida:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

Podemos reorganizar esse código escrevendo duas funções, em que cada uma executa uma tarefa específica. A maior parte do código não sofrerá alterações; estamos simplesmente deixando-o mais eficiente. A primeira função tratará a impressão dos designs e a segunda gerará um resumo da impressão feita:

```
u def print_models(unprinted_designs, completed_models):
      """
      Simula a impressão de cada design, até que não haja mais nenhum.
      Transfere cada design para completed_models após a impressão.
      """
      while unprinted_designs:
          current_design = unprinted_designs.pop()

          # Simula a criação de uma impressão 3D a partir do design
          print("Printing model: " + current_design)
          completed_models.append(current_design)

v def show_completed_models(completed_models):
      """Mostra todos os modelos impressos."""
      print("\nThe following models have been printed:")
      for completed_model in completed_models:
          print(completed_model)

  unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
  completed_models = []

  print_models(unprinted_designs, completed_models)
  show_completed_models(completed_models)
```

Em **u** definimos a função `print_models()` com dois parâmetros: uma lista de designs a serem impressos e uma lista de modelos concluídos. Dadas essas duas listas, a função simula a impressão de cada design esvaziando a lista de designs não impressos e preenchendo a lista de designs completos. Em **v** definimos a função `show_completed_models()` com um parâmetro: a lista de modelos finalizados. Dada essa lista, `show_completed_models()` exibe o nome de cada modelo impresso.

Esse programa produz a mesma saída da versão sem funções, mas o código está muito mais organizado. O código que faz a maior parte do

trabalho foi transferido para duas funções separadas, que deixam a parte principal do programa mais fácil de entender. Observe o corpo do programa para ver como é mais simples entender o que esse programa faz:

```
unprinted_designs = ['iphone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Definimos uma lista de designs não impressos e uma lista vazia que armazenará os modelos finalizados. Então, como já definimos nossas duas funções, tudo que temos a fazer é chamá-las e passar os argumentos corretos a elas. Chamamos `print_models()` e passamos as duas listas de que essa função precisa; conforme esperado, `print_models()` simula a impressão dos designs. Em seguida, chamamos `show_completed_models()` e passamos uma lista dos modelos concluídos para que ela possa apresentar os modelos impressos. Os nomes descritivos das funções permitem que outras pessoas leiam esse código e o entendam, mesmo que não haja comentários.

Esse programa é mais fácil de ser estendido e mantido que a versão sem funções. Se precisarmos imprimir mais designs depois, poderemos simplesmente chamar `print_models()` novamente. Se percebermos que o código de impressão precisa ser modificado, podemos alterar o código uma vez, e nossas alterações estarão presentes em todos os lugares em que a função for chamada. Essa técnica é mais eficiente que ter de atualizar um código separadamente em vários pontos do programa.

Esse exemplo também mostra a ideia de que toda função deve ter uma tarefa específica. A primeira função imprime cada design, enquanto a segunda mostra os modelos concluídos. Isso é mais vantajoso que usar uma única função para executar as duas tarefas. Se você estiver escrevendo uma função e perceber que ela está fazendo muitas tarefas diferentes, experimente dividir o código em duas funções. Lembre-se de que você sempre pode chamar uma função a partir de outras funções, o que pode ser conveniente quando dividimos uma tarefa complexa em uma série de passos.

Evitando que uma função modifique uma lista

Às vezes, você vai querer evitar que uma função modifique uma lista. Por exemplo, suponha que você comece com uma lista de designs não impressos e escreva uma função que transfira esses designs para uma lista de modelos terminados, como no exemplo anterior. Talvez você decida que, apesar de ter imprimido todos os designs, vai querer manter a lista original de designs não impressos em seus registros. Porém, como você transferiu todos os nomes de designs de `unprinted_designs`, a lista agora está vazia, e essa é a única versão da lista que você tem; a lista original se perdeu. Nesse caso, podemos tratar esse problema passando uma cópia da lista para a função, e não a lista original. Qualquer alteração que a função fizer na lista afetará apenas a cópia, deixando a lista original intacta.

Você pode enviar uma cópia de uma lista para uma função assim:

```
nome_da_função(nome_da_lista[:])
```

A notação de fatia `[:]` cria uma cópia da lista para ser enviada à função. Se não quiséssemos esvaziar a lista de designs não impressos em *print_models.py*, chamaríamos `print_models()` desta maneira:

```
print_models(unprinted_designs[:], completed_models)
```

A função `print_models()` pode fazer seu trabalho, pois ela continua recebendo os nomes de todos os designs não impressos. Porém, dessa vez, ela usa uma cópia da lista original de designs não impressos, e não a lista `unprinted_designs` propriamente dita. A lista `completed_models` será preenchida com os nomes dos modelos impressos, como antes, mas a lista original de designs não impressos não será afetada pela função.

Apesar de poder preservar o conteúdo de uma lista passando uma cópia dela para suas funções, você deve passar a lista original para as funções, a menos que tenha um motivo específico para passar uma cópia. Para uma função, é mais eficiente trabalhar com uma lista existente a fim de evitar o uso de tempo e de memória necessários para criar uma cópia separada, em especial quando trabalhamos com listas grandes.

FAÇA VOCÊ MESMO

8.9 – Mágicos: Crie uma lista de nomes de mágicos. Passe a lista para uma função chamada `show_magicians()` que exiba o nome de cada mágico da lista.

8.10 – Grandes mágicos: Comece com uma cópia de seu programa do Exercício 8.9. Escreva uma função chamada `make_great()` que modifique a lista de mágicos acrescentando a expressão *o Grande* ao nome de cada mágico. Chame `show_magicians()` para ver se a lista foi realmente modificada.

8.11 – Mágicos inalterados: Comece com o trabalho feito no Exercício 8.10. Chame a função `make_great()` com uma cópia da lista de nomes de mágicos. Como a lista original não será alterada, devolva a nova lista e armazene-a em uma lista separada. Chame `show_magicians()` com cada lista para mostrar que você tem uma lista de nomes originais e uma lista com a expressão *o Grande* adicionada ao nome de cada mágico.

Passando um número arbitrário de argumentos

Às vezes, você não saberá com antecedência quantos argumentos uma função deve aceitar. Felizmente, Python permite que uma função receba um número arbitrário de argumentos da instrução de chamada.

Por exemplo, considere uma função que prepare uma pizza. Ela deve aceitar vários ingredientes, mas não é possível saber com antecedência quantos ingredientes uma pessoa vai querer. A função no próximo exemplo tem um parâmetro `*toppings`, mas esse parâmetro agrupa tantos argumentos quantos forem fornecidos na linha de chamada:

pizza.py

```
def make_pizza(*toppings):
    """Exibe a lista de ingredientes pedidos."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

O asterisco no nome do parâmetro `*toppings` diz a Python para criar uma tupla vazia chamada `toppings` e reunir os valores recebidos nessa tupla. A instrução `print` no corpo da função gera uma saída que mostra que Python é capaz de tratar uma chamada de função com um valor e outra chamada com três valores. As chamadas são tratadas de modo semelhante. Observe

que Python agrupa os argumentos em uma tupla, mesmo que a função receba apenas um valor:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Podemos agora substituir a instrução `print` por um laço que percorra a lista de ingredientes e descreva a pizza sendo pedida:

```
def make_pizza(*toppings):
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

A função responde de forma apropriada, independentemente de receber um ou três valores:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Essa sintaxe funciona, não importa quantos argumentos a função receba.

Misturando argumentos posicionais e arbitrários

Se quiser que uma função aceite vários tipos de argumentos, o parâmetro que aceita um número arbitrário de argumentos deve ser colocado por último na definição da função. Python faz a correspondência de argumentos posicionais e nomeados antes, e depois agrupa qualquer argumento remanescente no último parâmetro.

Por exemplo, se a função tiver que aceitar um tamanho para a pizza, esse parâmetro deve estar antes do parâmetro `*toppings`:

```
def make_pizza(size, *toppings):
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Na definição da função, Python armazena o primeiro valor recebido no parâmetro `size`. Todos os demais valores que vierem depois são armazenados na tupla `toppings`. As chamadas da função incluem um argumento para o tamanho antes, seguido de tantos ingredientes quantos forem necessários.

Agora cada pizza tem um tamanho e alguns ingredientes, e cada informação é exibida no lugar apropriado, mostrando o tamanho antes e os ingredientes depois:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Usando argumentos nomeados arbitrários

Às vezes, você vai querer aceitar um número arbitrário de argumentos, mas não saberá com antecedência qual tipo de informação será passado para a função. Nesse caso, podemos escrever funções que aceitem tantos pares chave-valor quantos forem fornecidos pela instrução que faz a chamada. Um exemplo envolve criar perfis de usuários: você sabe que obterá informações sobre um usuário, mas não tem certeza quanto ao tipo de informação que receberá. A função `build_profile()` no próximo exemplo sempre aceita um primeiro nome e um sobrenome, mas aceita também um número arbitrário de argumentos nomeados:

user_profile.py

```
  def build_profile(first, last, **user_info):
      """Constrói um dicionário contendo tudo que sabemos sobre um
  usuário."""
      profile = {}
u     profile['first_name'] = first
      profile['last_name'] = last
v     for key, value in user_info.items():
          profile[key] = value
      return profile

  user_profile = build_profile('albert', 'einstein',
                               location='princeton',
                               field='physics')
  print(user_profile)
```

A definição de `build_profile()` espera um primeiro nome e um sobrenome e permite que o usuário passe tantos pares nome-valor quantos ele quiser. Os asteriscos duplos antes do parâmetro `**user_info` fazem Python criar um dicionário vazio chamado `user_info` e colocar quaisquer pares nome-valor recebidos nesse dicionário. Nessa função, podemos acessar os pares nome-valor em `user_info` como faríamos com qualquer dicionário.

No corpo de `build_profile()`, criamos um dicionário vazio chamado `profile` para armazenar o perfil do usuário. Em **u** adicionamos o primeiro nome e o sobrenome nesse dicionário porque sempre receberemos essas duas informações do usuário. Em **v** percorremos os pares chave-valor adicionais do dicionário `user_info` e adicionamos cada par ao dicionário `profile`. Por fim, devolvemos o dicionário `profile` à linha que chamou a função.

Chamamos `build_profile()` passando o primeiro nome `'albert'`, o sobrenome `'einstein'` e os dois pares chave-valor `location='princeton'` e `field='physics'`. Armazenamos o dicionário `profile` devolvido em `user_profile` e exibimos o valor dessa variável:

```
{'first_name': 'albert', 'last_name': 'einstein',
'location': 'princeton', 'field': 'physics'}
```

O dicionário devolvido contém o primeiro nome e o sobrenome do usuário e, nesse caso, a localidade e o campo de estudo também. A função será apropriada, não importa quantos pares chave-valor adicionais sejam fornecidos na chamada da função.

Podemos misturar valores posicionais, nomeados e arbitrários de várias maneiras diferentes quando escrevermos nossas próprias funções. É conveniente saber que todos esses tipos de argumento existem, pois você os verá com frequência quando começar a ler códigos de outras pessoas. Aprender a usar os diferentes tipos corretamente e saber quando usar cada um exige prática. Por enquanto, lembre-se de usar a abordagem mais simples possível, que faça o trabalho necessário. À medida que progredir, você aprenderá a usar a abordagem mais eficiente a cada vez.

FAÇA VOCÊ MESMO

8.12 – Sanduíches: Escreva uma função que aceite uma lista de itens que uma pessoa quer em um sanduíche. A função deve ter um parâmetro que agrupe tantos itens quantos forem fornecidos pela chamada da função e deve apresentar um resumo do sanduíche pedido. Chame a função três vezes usando um número diferente de argumentos a cada vez.

8.13 – Perfil do usuário: Comece com uma cópia de *user_profile.py,* da página 210. Crie um perfil seu chamando `build_profile()`, usando seu primeiro nome e o sobrenome, além de três outros pares chave-valor que o descrevam.

8.14 – Carros: Escreva uma função que armazene informações sobre um carro em um dicionário. A função sempre deve receber o nome de um fabricante e um modelo. Um número arbitrário de argumentos nomeados então deverá ser aceito. Chame a função com as informações necessárias e dois outros pares nome-valor, por exemplo, uma cor ou um opcional. Sua função deve ser apropriada para uma chamada como esta:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Mostre o dicionário devolvido para garantir que todas as informações foram armazenadas corretamente.

Armazenando suas funções em módulos

Uma vantagem das funções é a maneira como elas separam blocos de código de seu programa principal. Ao usar nomes descritivos para suas

funções, será bem mais fácil entender o seu programa principal. Você pode dar um passo além armazenando suas funções em um arquivo separado chamado *módulo* e, então, *importar* esse módulo em seu programa principal. Uma instrução `import` diz a Python para deixar o código de um módulo disponível no arquivo de programa em execução no momento.

Armazenar suas funções em um arquivo separado permite ocultar os detalhes do código de seu programa e se concentrar na lógica de nível mais alto. Também permite reutilizar funções em muitos programas diferentes. Quando armazenamos funções em arquivos separados, podemos compartilhar esses arquivos com outros programadores sem a necessidade de compartilhar o programa todo. Saber como importar funções também possibilita usar bibliotecas de funções que outros programadores escreveram.

Há várias maneiras de importar um módulo e vou mostrar cada uma delas rapidamente.

Importando um módulo completo

Para começar a importar funções, inicialmente precisamos criar um módulo. Um *módulo* é um arquivo terminado em *.py* que contém o código que queremos importar para o nosso programa. Vamos criar um módulo que contenha a função `make_pizza()`. Para criar esse módulo, removeremos tudo que está no arquivo *pizza.py*, exceto a função `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)
```

Agora criaremos um arquivo separado chamado *making_pizzas.py* no mesmo diretório em que está *pizza.py*. Esse arquivo importa o módulo que acabamos de criar e, em seguida, faz duas chamadas para `make_pizza()`:

making_pizzas.py

```
  import pizza

u pizza.make_pizza(16, 'pepperoni')
  pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Quando Python lê esse arquivo, a linha `import pizza` lhe diz para abrir o arquivo *pizza.py* e copiar todas as funções dele para esse programa. Você não vê realmente o código sendo copiado entre os arquivos porque Python faz isso internamente, à medida que o programa executa. Tudo que você precisa saber é que qualquer função definida em *pizza.py* agora estará disponível em *making_pizzas.py*.

Para chamar uma função que está em um módulo importado, forneça o nome do módulo, que é `pizza` nesse caso, seguido do nome da função, `make_pizza()`, separados por um ponto u. Esse código gera a mesma saída que o programa original, que não importava um módulo:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Essa primeira abordagem à importação, em que simplesmente escrevemos `import` seguido do nome do módulo, deixa todas as funções do módulo disponíveis ao seu programa. Se você usar esse tipo de instrução `import` para importar um módulo completo chamado *nome_do_módulo.py*, todas as funções do módulo estarão disponíveis por meio da sintaxe a seguir:

```
nome_do_módulo.nome_da_função()
```

Importando funções específicas

Podemos também importar uma função específica de um módulo. Eis a sintaxe geral para essa abordagem:

```
from nome_do_módulo import nome_da_função
```

Você pode importar quantas funções quiser de um módulo separando o nome de cada função com uma vírgula:

```
from nome_do_módulo import função_0, função_1, função_2
```

O exemplo com *making_pizzas.py* teria o seguinte aspecto, se quiséssemos importar somente a função que será utilizada:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Com essa sintaxe não precisamos usar a notação de ponto ao chamar uma função. Como importamos explicitamente a função `make_pizza()` na instrução `import`, podemos chamá-la pelo nome quando ela for utilizada.

Usando a palavra reservada as para atribuir um alias a uma função

Se o nome de uma função que você importar puder entrar em conflito com um nome existente em seu programa ou se o nome da função for longo, podemos usar um *alias* único e conciso, que é um nome alternativo, semelhante a um apelido para a função. Dê esse apelido especial à função quando importá-la.

A seguir, atribuímos um alias `mp()` para a função `make_pizza()` importando `make_pizza as mp`. A palavra reservada `as` renomeia uma função usando o alias que você fornecer:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

A instrução `import` no exemplo renomeia a função `make_pizza()` para `mp()` nesse programa. Sempre que quiser chamar `make_pizza()`, você pode simplesmente escrever `mp()` em seu lugar, e Python executará o código de `make_pizza()`, ao mesmo tempo que evitará confusão com outra função `make_pizza()` que você possa ter escrito nesse arquivo de programa.

A sintaxe geral para fornecer um alias é:

```
from nome_do_módulo import nome_da_função as nf
```

Usando a palavra reservada as para atribuir um alias a um módulo

Também podemos fornecer um alias para um nome de módulo. Dar um alias conciso a um módulo, por exemplo, `p` para `pizza`, permite chamar mais rapidamente as funções do módulo. Chamar `p.make_pizza()` é mais compacto que chamar `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

O módulo `pizza` recebe o alias `p` na instrução `import`, mas todas as funções do módulo preservam seus nomes originais. Chamar as funções escrevendo `p.make_pizza()` não só é mais compacto que escrever `pizza.make_pizza()` como também desvia sua atenção do nome do módulo e permite dar enfoque aos nomes descritivos de suas funções. Esses nomes de função, que claramente informam o que cada função faz, são mais importantes para a legibilidade de seu código que usar o nome completo do módulo.

A sintaxe geral para essa abordagem é:

```
import nome_do_módulo as nm
```

Importando todas as funções de um módulo

Podemos dizer a Python para importar todas as funções de um módulo usando o operador asterisco (`*`):

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

O asterisco na instrução `import` diz a Python para copiar todas as funções do módulo `pizza` para esse arquivo de programa. Como todas as funções são importadas, podemos chamar qualquer função pelo nome, sem usar a notação de ponto. No entanto, é melhor não usar essa abordagem quando trabalhar com módulos maiores, que não foram escritos por você: se o

módulo tiver um nome de função que seja igual a um nome existente em seu projeto, você poderá ter alguns resultados inesperados. Python poderá ver várias funções ou variáveis com o mesmo nome e, em vez de importar todas as funções separadamente, ele as sobrescreverá.

A melhor abordagem é importar a função ou as funções que você quiser ou importar o módulo todo e usar a notação de ponto. Isso resulta em um código claro, mais fácil de ler e de entender. Incluí esta seção para que você possa reconhecer instruções `import` como esta quando as vir no código de outras pessoas:

```
from nome_do_módulo import *
```

Estilizando funções

Você precisa ter alguns detalhes em mente quando estilizar funções. As funções devem ter nomes descritivos, e esses nomes devem usar letras minúsculas e underscores. Nomes descritivos ajudam você e outras pessoas a entenderem o que seu código está tentando fazer. Os nomes de módulos devem usar essas convenções também.

Toda função deve ter um comentário que explique o que ela faz de modo conciso. Esse comentário deve estar imediatamente após a definição da função e deve utilizar o formato de docstring. Se uma função estiver bem documentada, outros programadores poderão usá-la apenas lendo a descrição da docstring. Eles poderão crer que o código funcionará conforme descrito e, desde que saibam o nome da função, os argumentos necessários e o tipo de valor que ela devolve, deverão ser capazes de usá-la em seus programas.

Se você especificar um valor default para um parâmetro, não deve haver espaços em nenhum dos lados do sinal de igualdade:

```
def nome_da_função(parâmetro_0, parâmetro_1='valor default')
```

A mesma convenção deve ser usada para argumentos nomeados em chamadas de função:

```
nome_da_função(valor_0, parâmetro_1='valor')
```

A PEP 8 (*https://www.python.org/dev/peps/pep-0008/*) recomenda limitar as linhas de código em 79 caracteres para que todas as linhas permaneçam visíveis em uma janela de editor com um tamanho razoável. Se um conjunto de parâmetros fizer com que a definição de uma função ultrapasse 79 caracteres, tecle ENTER após o parêntese de abertura na linha da definição. Na próxima linha, tecle TAB duas vezes para separar a lista de argumentos do corpo da função, que estará indentado com apenas um nível.

A maioria dos editores fará automaticamente o alinhamento de qualquer parâmetro adicional para que corresponda à indentação que você determinar na primeira linha:

```
def nome_da_função(
        parâmetro_0, parâmetro_1, parâmetro_2,
        parâmetro_3, parâmetro_4, parâmetro_5):
    corpo da função...
```

Se seu programa ou módulo tiver mais de uma função, você poderá separá-las usando duas linhas em branco para facilitar ver em que lugar uma função termina e a próxima começa.

Todas as instruções `import` devem estar no início de um arquivo. A única exceção ocorre quando você usa comentários no início de seu arquivo para descrever o programa como um todo.

FAÇA VOCÊ MESMO

8.15 – Impressão de modelos: Coloque as funções do exemplo *print_models.py* em um arquivo separado de nome *printing_functions.py*. Escreva uma instrução `import` no início de *print_models.py* e modifique o arquivo para usar as funções importadas.

8.16 – Importações: Usando um programa que você tenha escrito e que contenha uma única função, armazene essa função em um arquivo separado. Importe a função para o arquivo principal de seu programa e chame-a usando cada uma das seguintes abordagens:

```
import nome_do_módulo
from nome_do_módulo import nome_da_função
from nome_do_módulo import nome_da_função as nf
```

```
import nome_do_módulo as nm
from nome_do_módulo import *
```

8.17 – Estilizando funções: Escolha quaisquer três programas que você escreveu neste capítulo e garanta que estejam de acordo com as diretrizes de estilo descritas nesta seção.

Resumo

Neste capítulo aprendemos a escrever funções e a passar argumentos de modo que suas funções tenham acesso às informações necessárias para realizarem sua tarefa. Vimos como usar argumentos posicionais e nomeados, e aprendemos a aceitar um número arbitrário de argumentos. Conhecemos funções que exibem uma saída e funções que devolvem valores. Aprendemos a usar funções com listas, dicionários, instruções `if` e laços `while`. Também vimos como armazenar suas funções em arquivos separados chamados de *módulos*; desse modo, seus arquivos de programa serão mais simples e mais fáceis de entender. Por fim, aprendemos a estilizar as funções para que seus programas continuem bem estruturados e o mais fácil possível para você e outras pessoas lerem.

Um de nossos objetivos como programador deve ser escrever um código simples, que faça o que você quer, e as funções ajudam nessa tarefa. Elas permitem escrever blocos de código e deixá-los de lado depois que você souber que eles funcionam. Quando souber que uma função executa sua tarefa de forma correta, você poderá se sentir seguro de que ela continuará a funcionar, e poderá prosseguir para a próxima tarefa de programação.

As funções permitem escrever um código uma vez e então reutilizá-lo quantas vezes você quiser. Quando tiver que executar o código de uma função, tudo que você precisa fazer é escrever uma chamada de uma linha, e a função fará o seu trabalho. Quando for necessário modificar o comportamento de uma função, basta modificar apenas um bloco de código e sua alteração terá efeito em todos os lugares em que uma chamada dessa função for feita.

Usar funções deixa seus programas mais fáceis de ler, e bons nomes de função sintetizam o que cada parte de um programa faz. Ler uma série de chamadas de função possibilita ter rapidamente uma noção do que um

programa faz, se comparado à leitura de uma série longa de blocos de código.

As funções também deixam seu código mais fácil de testar e de depurar. Quando a maior parte do trabalho de seu programa for feita por um conjunto de funções, em que cada uma tem uma tarefa específica, será muito mais fácil testar e dar manutenção no código que você escreveu. É possível escrever um programa separado que chame cada função e testar se ela funciona em todas as situações com as quais você poderá se deparar. Ao fazer isso, você se sentirá mais seguro de que suas funções estarão corretas sempre que forem chamadas.

No Capítulo 9 aprenderemos a escrever classes. As classes combinam funções e dados em um pacote organizado, que pode ser usado de maneiras flexíveis e eficientes.

9

CLASSES

A *programação orientada a objetos* é uma das abordagens mais eficientes para escrever software. Na programação orientada a objetos, escrevemos *classes* que representam entidades e situações do mundo real, e criamos *objetos* com base nessas classes. Quando escrevemos uma classe, definimos o comportamento geral que toda uma categoria de objetos pode ter.

Quando criamos objetos individuais a partir da classe, cada objeto será automaticamente equipado com o comportamento geral; então você poderá dar a cada objeto as características únicas que desejar. Você ficará impressionado ao ver como situações do mundo real podem ser bem modeladas com a programação orientada a objetos.

Criar um objeto a partir de uma classe é uma operação conhecida como *instanciação*, e trabalhamos com *instâncias* de uma classe. Neste capítulo, escreveremos classes e criaremos instâncias dessas classes. Especificaremos o tipo de informação que pode ser armazenado nas instâncias e definiremos ações que podem ser executadas nessas instâncias. Também escreveremos classes que estendem a funcionalidade de classes existentes, de modo que classes semelhantes possam compartilhar códigos de forma eficiente. Armazenaremos nossas classes em módulos e importaremos classes escritas por outros programadores para nossos próprios arquivos de programa.

Entender a programação orientada a objetos ajudará a ver o mundo como um programador o vê. Ela ajudará você a realmente conhecer o seu código, não apenas o que acontece linha a linha, mas também os conceitos mais amplos por trás dele. Conhecer a lógica por trás das classes treinará você a pensar de modo lógico a fim de poder escrever programas que tratem praticamente todo problema encontrado de forma eficiente.

As classes também facilitam sua vida e a vida de outros programadores com quem você precisará trabalhar à medida que assumir desafios cada vez

mais complexos. Quando você e outros programadores escreverem códigos baseados no mesmo tipo de lógica, vocês serão capazes de entender o trabalho uns dos outros. Seus programas farão sentido para muitos colaboradores, permitindo que todos façam mais.

Criando e usando uma classe

Podemos modelar de tudo usando classes. Vamos começar escrevendo uma classe simples, `Dog`, que representa um cachorro – não um cachorro em particular, mas qualquer cachorro. O que sabemos sobre a maioria dos cachorros de estimação? Bem, todos eles têm um nome e uma idade. Também sabemos que a maioria deles senta e rola. Essas duas informações (nome e idade) e esses dois comportamentos (sentar e rolar) farão parte de nossa classe `Dog`, pois são comuns à maioria dos cachorros. Essa classe dirá à Python como criar um objeto que represente um cachorro. Depois que nossa classe estiver escrita, ela será usada para criar instâncias individuais, em que cada uma representará um cachorro específico.

Criando a classe Dog

Cada instância criada a partir da classe `Dog` armazenará um nome (`name`) e uma idade (`age`), e daremos a cada cachorro a capacidade de sentar (`sit()`) e rolar (`roll_over()`):

dog.py

```
u class Dog():
v     """Uma tentativa simples de modelar um cachorro."""

w     def __init__(self, name, age):
          """Inicializa os atributos name e age."""
x         self.name = name
          self.age = age

y     def sit(self):
          """Simula um cachorro sentando em resposta a um comando."""
          print(self.name.title() + " is now sitting.")
```

```
    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando."""
        print(self.name.title() + " rolled over!")
```

Há vários aspectos a serem observados aqui, mas não se preocupe. Você verá essa estrutura ao longo deste capítulo e terá bastante tempo para se acostumar com ela. Em u definimos uma classe chamada Dog. Por convenção, nomes com a primeira letra maiúscula referem-se a classes em Python. Os parênteses na definição da classe estão vazios porque estamos criando essa classe do zero. Em v escrevemos uma docstring que descreve o que essa classe faz.

Método __init__()

Uma função que faz parte de uma classe é um *método*. Tudo que aprendemos sobre funções também se aplica aos métodos; a única diferença prática, por enquanto, é o modo como chamaremos os métodos. O método __init__() em w é um método especial que Python executa automaticamente sempre que criamos uma nova instância baseada na classe Dog. Esse método tem dois underscores no início e dois no final – uma convenção que ajuda a evitar que os nomes default de métodos Python entrem em conflito com nomes de métodos criados por você.

Definimos o método __init__() para que tenha três parâmetros: self, name e age. O parâmetro self é obrigatório na definição do método e deve estar antes dos demais parâmetros. Deve estar incluído na definição, pois, quando Python chama esse método __init__() depois (para criar uma instância de Dog), a chamada do método passará o argumento self automaticamente. Toda chamada de método associada a uma classe passa self, que é uma referência à própria instância, de modo automático; ele dá acesso aos atributos e métodos da classe à instância individual. Quando criamos uma instância de Dog, Python chamará o método __init__() da classe Dog. Passaremos um nome e uma idade como argumentos para Dog(); self é passado automaticamente, portanto não é preciso especificá-lo. Sempre que quisermos criar uma instância da classe Dog forneceremos valores apenas para os dois últimos parâmetros, que são name e age.

As duas variáveis definidas em x têm o prefixo self. Qualquer variável

prefixada com `self` está disponível a todos os métodos da classe; além disso, podemos acessar essas variáveis por meio de qualquer instância criada a partir da classe. `self.name = name` usa o valor armazenado no parâmetro `name` e o armazena na variável `name`, que é então associada à instância criada. O mesmo processo ocorre com `self.age = age`. Variáveis como essas, acessíveis por meio de instâncias, são chamadas de *atributos*.

A classe `Dog` tem dois outros dois métodos definidos: `sit()` e `roll_over()` `y`. Como esses métodos não precisam de informações adicionais como um nome ou uma idade, simplesmente os definimos com um parâmetro `self`. As instâncias que criarmos posteriormente terão acesso a esses métodos. Em outras palavras, elas terão a capacidade de sentar e rolar. Por enquanto, `sit()` e `roll_over()` não fazem muito. Apenas exibem uma mensagem dizendo que o cachorro está sentando ou rolando. No entanto, o conceito pode ser estendido para situações realistas: se essa classe fizesse parte de um jogo de computador de verdade, esses métodos conteriam código para fazer a animação de um cachorro sentando e rolando. Se essa classe tivesse sido escrita para controlar um robô, esses métodos direcionariam os movimentos para fazer um cachorro-robô sentar e rolar.

Criando classes em Python 2.7

Quando criar uma classe em Python 2.7, será necessário fazer uma pequena mudança. Inclua o termo `object` entre parênteses quando criá-la:

```
class NomeClasse(object):
    --trecho omitido--
```

Isso faz as classes de Python 2.7 se comportarem de modo mais próximo das classes de Python 3, o que, de modo geral, tornará seu trabalho mais simples.

A classe `Dog` seria definida assim em Python 2.7:

```
class Dog(object):
    --trecho omitido--
```

Criando uma instância a partir de uma classe

Pense em uma classe como um conjunto de instruções para criar uma

instância. A classe Dog é um conjunto de instruções que diz a Python como criar instâncias individuais que representem cachorros específicos.

Vamos criar uma instância que represente um cachorro específico:

```
class Dog():
    --trecho omitido--

u my_dog = Dog('willie', 6)

v print("My dog's name is " + my_dog.name.title() + ".")
w print("My dog is " + str(my_dog.age) + " years old.")
```

A classe Dog que usamos aqui é aquela que acabamos de escrever no exemplo anterior. Em u dizemos a Python para criar um cachorro de nome 'willie' e idade igual a 6. Quando Python lê essa linha, ele chama o método __init__() de Dog com os argumentos 'willie' e 6. O método __init__() cria uma instância que representa esse cachorro em particular e define os atributos name e age com os valores que fornecemos. Esse método não tem uma instrução return explícita, mas Python devolve automaticamente uma instância que representa esse cachorro. Armazenamos essa instância na variável my_dog. A convenção de nomenclatura é útil nesse caso: em geral, podemos supor que um nome com a primeira letra maiúscula como Dog refere-se a uma classe, enquanto um nome com letras minúsculas como my_dog refere-se a uma única instância criada a partir de uma classe.

Acessando atributos

Para acessar os atributos de uma instância utilize a notação de ponto. Em v acessamos o valor do atributo name de my_dog escrevendo:

```
my_dog.name
```

A notação de ponto é usada com frequência em Python. Essa sintaxe mostra como Python encontra o valor de um atributo. Nesse caso, o interpretador olha para a instância my_dog e encontra o atributo name associado a ela. É o mesmo atributo referenciado como self.name na classe Dog. Em w usamos a mesma abordagem para trabalhar com o atributo age.

Em nossa primeira instrução `print`, `my_dog.name.title()` faz com que `'willie'` – o valor do atributo `name` de `my_dog` – comece com uma letra maiúscula. Na segunda instrução `print`, `str(my_dog.age)` converte `6` – o valor do atributo `age` de `my_dog` – em uma string.

A saída é um resumo do que sabemos sobre `my_dog`:

```
My dog's name is Willie.
My dog is 6 years old.
```

Chamando métodos

Depois que criarmos uma instância da classe `Dog`, podemos usar a notação de ponto para chamar qualquer método definido nessa classe. Vamos fazer nosso cachorro sentar e rolar:

```
class Dog():
    --trecho omitido--

my_dog = Dog('willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Para chamar um método, especifique o nome da instância (nesse caso, `my_dog`) e o método que você quer chamar, separados por um ponto. Quando Python lê `my_dog.sit()`, ele procura o método `sit()` na classe `Dog` e executa esse código. A linha `my_dog.roll_over()` é interpretada do mesmo modo.

Agora Willie faz o que lhe dissemos:

```
Willie is now sitting.
Willie rolled over!
```

Essa sintaxe é bem conveniente. Quando atributos e métodos recebem nomes descritivos de forma apropriada, por exemplo, `name`, `age`, `sit()` e `roll_over()`, podemos inferir facilmente o que um bloco de código deve fazer, mesmo um que nunca vimos antes.

Criando várias instâncias

Você pode criar tantas instâncias de uma classe quantas forem necessárias.

Vamos criar um segundo cachorro chamado `your_dog`:

```
class Dog():
    --trecho omitido--

my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
my_dog.sit()

print("\nYour dog's name is " + your_dog.name.title() + ".")
print("Your dog is " + str(your_dog.age) + " years old.")
your_dog.sit()
```

Nesse exemplo, criamos um cachorro chamado Willie e uma cadela de nome Lucy. Cada cachorro é uma instância separada, com seu próprio conjunto de atributos, capaz de realizar o mesmo conjunto de ações:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Mesmo que usássemos o mesmo nome e a mesma idade para o segundo cachorro, Python criaria uma instância separada da classe `Dog`. Você pode criar tantas instâncias de uma classe quantas forem necessárias, desde que dê a cada instância um nome de variável único ou que ela ocupe uma única posição em uma lista ou dicionário.

FAÇA VOCÊ MESMO

9.1 – Restaurante: Crie uma classe chamada `Restaurant`. O método `__init__()` de `Restaurant` deve armazenar dois atributos: `restaurant_name` e `cuisine_type`. Crie um método chamado `describe_restaurant()` que mostre essas duas informações, e um método de nome `open_restaurant()` que exiba uma

mensagem informando que o restaurante está aberto.

Crie uma instância chamada `restaurant` a partir de sua classe. Mostre os dois atributos individualmente e, em seguida, chame os dois métodos.

9.2 – Três restaurantes: Comece com a classe do Exercício 9.1. Crie três instâncias diferentes da classe e chame `describe_restaurant()` para cada instância.

9.3 – Usuários: Crie uma classe chamada `User`. Crie dois atributos de nomes `first_name` e `last_name` e, então, crie vários outros atributos normalmente armazenados em um perfil de usuário. Escreva um método de nome `describe_user()` que apresente um resumo das informações do usuário. Escreva outro método chamado `greet_user()` que mostre uma saudação personalizada ao usuário.

Crie várias instâncias que representem diferentes usuários e chame os dois métodos para cada usuário.

Trabalhando com classes e instâncias

Podemos usar classes para representar muitas situações do mundo real. Depois que escrever uma classe, você gastará a maior parte de seu tempo trabalhando com instâncias dessa classe. Uma das primeiras tarefas que você vai querer fazer é modificar os atributos associados a uma instância em particular. Podemos modificar os atributos de uma instância diretamente, ou escrever métodos que atualizem os atributos de formas específicas.

Classe Car

Vamos escrever uma nova classe que represente um carro. Nossa classe armazenará informações sobre o tipo de carro com que estamos trabalhando e terá um método que sintetiza essa informação:

car.py

```
class Car():
    """Uma tentativa simples de representar um carro."""

u     def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
```

```
        self.model = model
        self.year = year

v     def get_descriptive_name(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

w my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car.get_descriptive_name())
```

Na classe `Car` em u definimos o método `__init__()` com o parâmetro `self` em primeiro lugar, exatamente como fizemos antes com nossa classe `Dog`. Também fornecemos outros três parâmetros: `make`, `model` e `year`. O método `__init__()` aceita esses parâmetros e os armazena nos atributos que serão associados às instâncias criadas a partir dessa classe. Quando criarmos uma nova instância de `Car`, precisaremos especificar um fabricante, um modelo e o ano para a nossa instância.

Em v definimos um método chamado `get_descriptive_name()` que coloca os atributos `year`, `make` e `model` de um carro em uma string, descrevendo o carro de modo elegante. Isso evitará a necessidade de exibir o valor de cada atributo individualmente. Para trabalhar com os valores dos atributos nesse método, usamos `self.make`, `self.model` e `self.year`. Em w criamos uma instância da classe `Car` e a armazenamos na variável `my_new_car`. Então chamamos `get_descriptive_name()` para mostrar o tipo de carro que temos:

```
2016 Audi A4
```

Para deixar a classe mais interessante, vamos adicionar um atributo que mude com o tempo. Acrescentaremos um atributo que armazena a milhagem do carro.

Definindo um valor default para um atributo

Todo atributo de uma classe precisa de um valor inicial, mesmo que esse valor seja 0 ou uma string vazia. Em alguns casos, por exemplo, quando definimos um valor default, faz sentido especificar esse valor inicial no corpo do método `__init__()`; se isso for feito para um atributo, você não

precisará incluir um parâmetro para ele.

Vamos acrescentar um atributo chamado `odometer_reading` que sempre começa com o valor 0. Também adicionaremos um método `read_odometer()` que nos ajudará a ler o hodômetro de cada carro:

```
class Car():

    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year
u       self.odometer_reading = 0

    def get_descriptive_name(self):
        --trecho omitido--

v   def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on
it.")

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Dessa vez, quando Python chama o método `__init__()` para criar uma nova instância, os valores para o fabricante, o modelo e o ano são armazenados como atributos, como fizemos no exemplo anterior. Em seguida, Python cria um novo atributo chamado `odometer_reading` e define seu valor inicial com 0 u. Também temos um novo método de nome `read_odometer()` em v que facilita a leitura da milhagem de um carro.

Nosso carro tem a milhagem iniciada com 0:

```
2016 Audi A4
This car has 0 miles on it.
```

Não há muitos carros vendidos com o hodômetro marcando exatamente 0, portanto precisamos de uma maneira de alterar o valor desse atributo.

Modificando valores de atributos

Você pode alterar o valor de um atributo de três maneiras: podemos modificar o valor diretamente por meio de uma instância, definir o valor com um método ou incrementá-lo (somar um determinado valor a ele). Vamos analisar cada uma dessas abordagens.

Modificando o valor de um atributo diretamente

A maneira mais simples de modificar o valor de um atributo é acessá-lo diretamente por meio de uma instância. A seguir, definimos o valor de leitura do hodômetro para 23, de forma direta:

```
  class Car():
      --trecho omitido--

  my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car.get_descriptive_name())

u my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

Em u usamos a notação de ponto para acessar o atributo `odometer_reading` do carro e definir seu valor diretamente. Essa linha diz a Python para usar a instância `my_new_car`, encontrar o atributo `odometer_reading` associado a ela e definir o valor desse atributo com 23:

```
2016 Audi A4
This car has 23 miles on it.
```

Às vezes, você vai querer acessar os atributos de forma direta como fizemos, mas, em outras ocasiões, vai querer escrever um método que atualize o valor para você.

Modificando o valor de um atributo com um método

Pode ser conveniente ter métodos que atualizem determinados atributos para você. Em vez de acessar o atributo de modo direto, passe o novo valor para um método que trate a atualização internamente.

Eis um exemplo que mostra um método de nome `update_odometer()`:

```
class Car():
    --trecho omitido--

u     def update_odometer(self, mileage):
        """Define o valor de leitura do hodômetro com o valor
especificado."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

v my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

A única modificação em `Car` foi o acréscimo de `update_odometer()` em **u**. Esse método aceita um valor de milhagem e o armazena em `self.odometer_reading`. Em **v** chamamos `update_odometer()` e lhe passamos o valor `23` como argumento (correspondendo ao parâmetro `mileage` na definição do método). Esse método define o valor de leitura do hodômetro com 23 e `read_odometer()` exibe essa leitura:

```
2016 Audi A4
This car has 23 miles on it.
```

Podemos estender o método `update_odometer()` para que faça uma tarefa adicional sempre que a leitura do hodômetro for modificada. Vamos acrescentar um pouco de lógica para garantir que ninguém tente diminuir o valor lido no hodômetro:

```
class Car():
    --trecho omitido--

    def update_odometer(self, mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado
        Rejeita a alteração se for tentativa de definir um valor menor
para o hodômetro
        """
u         if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
```

```
v             print("You can't roll back an odometer!")
```

Agora `update_odometer()` verifica se o novo valor do hodômetro faz sentido antes de modificar o atributo. Se a nova milhagem, `mileage`, for maior ou igual à milhagem existente, `self.odometer_reading`, você poderá atualizar o valor de leitura do hodômetro com a nova milhagem **u**. Se a nova milhagem for menor que a milhagem existente, você receberá um aviso informando que não pode diminuir o valor lido no hodômetro **v**.

Incrementando o valor de um atributo com um método

Às vezes, você vai querer incrementar o valor de um atributo de determinada quantidade, em vez de definir um valor totalmente novo. Suponha que compramos um carro usado e andamos cem milhas entre o instante em que o compramos e o momento em que o registramos. Eis um método que nos permite passar essa quantidade incremental e somar esse valor ao valor de leitura do hodômetro:

```
class Car():
    --trecho omitido--

    def update_odometer(self, mileage):
        --trecho omitido--

u     def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do
hodômetro."""
        self.odometer_reading += miles

v my_used_car = Car('subaru', 'outback', 2013)
print(my_used_car.get_descriptive_name())

w my_used_car.update_odometer(23500)
my_used_car.read_odometer()

x my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

O novo método `increment_odometer()` em **u** aceita uma quantidade de milhas e soma esse valor a `self.odometer_reading`. Em **v** criamos um carro

usado, `my_used_car`. Definimos seu hodômetro com o valor 23.500 chamando `update_odometer()` e passando-lhe o valor `23500` em **w**. Em **x** chamamos `increment_odometer()` e passamos o valor `100` para somar as cem milhas que dirigimos entre comprar o carro e registrá-lo:

```
2013 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

Você pode modificar facilmente esse método para rejeitar incrementos negativos, de modo que ninguém possa usar essa função para reduzir o valor lido no hodômetro.

NOTA Podemos usar métodos como esse para controlar o modo como os usuários de seu programa atualizam informações como o valor de leitura do hodômetro, mas qualquer pessoa com acesso ao programa poderá definir o valor do hodômetro com um valor qualquer acessando o atributo diretamente. Uma segurança eficiente exige atenção extrema aos detalhes, além de verificações básicas como essas mostradas aqui.

FAÇA VOCÊ MESMO

9.4 – Pessoas atendidas: Comece com seu programa do Exercício 9.1 (página 225). Acrescente um atributo chamado `number_served` cujo valor default é 0. Crie uma instância chamada `restaurant` a partir dessa classe. Apresente o número de clientes atendidos pelo restaurante e, em seguida, mude esse valor e exiba-o novamente.

Adicione um método chamado `set_number_served()` que permita definir o número de clientes atendidos. Chame esse método com um novo número e mostre o valor novamente.

Acrescente um método chamado `increment_number_served()` que permita incrementar o número de clientes servidos. Chame esse método com qualquer número que você quiser e que represente quantos clientes foram atendidos, por exemplo, em um dia de funcionamento.

9.5 – Tentativas de login: Acrescente um atributo chamado `login_attempts` à sua classe `User` do Exercício 9.3 (página 226). Escreva um método chamado `increment_login_attempts()` que incremente o valor de `login_attempts` em 1. Escreva outro método chamado `reset_login_attempts()` que reinicie o valor de `login_attempts` com 0.

Crie uma instância da classe `User` e chame `increment_login_attempts()`

várias vezes. Exiba o valor de `login_attempts` para garantir que ele foi incrementado de forma apropriada e, em seguida, chame `reset_login_attempts()`. Exiba `login_attempts` novamente para garantir que seu valor foi reiniciado com 0.

Herança

Nem sempre você precisará começar do zero para escrever uma classe. Se a classe que você estiver escrevendo for uma versão especializada de outra classe já criada, a *herança* poderá ser usada. Quando uma classe *herda* de outra, ela assumirá automaticamente todos os atributos e métodos da primeira classe. A classe original se chama *classe-pai* e a nova classe é a *classe-filha*. A classe-filha herda todos os atributos e método de sua classe-pai, mas também é livre para definir novos atributos e métodos próprios.

Método __init__() de uma classe-filha

A primeira tarefa de Python ao criar uma instância de uma classe-filha é atribuir valores a todos os atributos da classe-pai. Para isso, o método `__init__()` de uma classe-filha precisa da ajuda de sua classe-pai.

Como exemplo, vamos modelar um carro elétrico. Um carro elétrico é apenas um tipo específico de carro, portanto podemos basear nossa nova classe `ElectricCar` na classe `Car` que escrevemos antes. Então só precisaremos escrever código para os atributos e os comportamentos específicos de carros elétricos.

Vamos começar criando uma versão simples da classe `ElectricCar` que faz tudo que a classe `Car` faz:

electric_car.py

```
u class Car():
      """Uma tentativa simples de representar um carro."""

      def __init__(self, make, model, year):
          self.make = make
          self.model = model
          self.year = year
          self.odometer_reading = 0
```

```
    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        print("This car has " + str(self.odometer_reading) + " miles on
it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

v class ElectricCar(Car):
    """Representa aspectos específicos de veículos elétricos."""

w    def __init__(self, make, model, year):
        """Inicializa os atributos da classe-pai."""
x        super().__init__(make, model, year)


y my_tesla = ElectricCar('tesla', 'model s', 2016)
 print(my_tesla.get_descriptive_name())
```

Em `u` começamos com `Car`. Quando criamos uma classe-filha, a classe-pai deve fazer parte do arquivo atual e deve aparecer antes da classe-filha no arquivo. Em `v` definimos a classe-filha, que é `ElectricCar`. O nome da classe-pai deve ser incluído entre parênteses na definição da classe-filha. O método `__init__()` em `w` aceita as informações necessárias para criar uma instância de `Car`.

A função `super()` em `x` é uma função especial que ajuda Python a criar conexões entre a classe-pai e a classe-filha. Essa linha diz a Python para chamar o método `__init__()` da classe-pai de `ElectricCar`, que confere todos os atributos da classe-pai a `ElectricCar`. O nome *super* é derivado de

uma convenção segundo a qual a classe-pai se chama *superclasse* e a classe-filha é a *subclasse*.

Testamos se a herança está funcionando de forma apropriada tentando criar um carro elétrico com o mesmo tipo de informação que fornecemos quando criamos um carro comum. Em y criamos uma instância da classe `ElectricCar` e a armazenamos em `my_tesla`. Essa linha chama o método `__init__()` definido em `ElectricCar` que, por sua vez, diz a Python para chamar o método `__init__()` definido na classe-pai `Car`. Fornecemos os argumentos `'tesla'`, `'model s'` e `2016`.

Além de `__init__()` não há outros atributos nem métodos que sejam particulares a um carro elétrico. A essa altura, estamos apenas garantindo que o carro elétrico tenha o comportamento apropriado de `Car`:

```
2016 Tesla Model S
```

A instância de `ElectricCar` funciona exatamente como uma instância de `Car`, portanto podemos agora começar a definir atributos e métodos específicos aos carros elétricos.

Herança em Python 2.7

Em Python 2.7, a herança é um pouco diferente. A classe `ElectricCar` teria o seguinte aspecto:

```
class Car(object):
    def __init__(self, make, model, year):
        --trecho omitido--

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(make, model, year)
        --trecho omitido--
```

A função `super()` precisa de dois argumentos: uma referência à classe-filha e o objeto `self`. Esses argumentos são necessários para ajudar Python a fazer as conexões apropriadas entre as classes pai e filha. Quando usar herança em Python 2.7, lembre-se de definir a classe-pai usando a sintaxe `object` também.

Definindo atributos e métodos da classe-filha

Depois que tiver uma classe-filha que herde de uma classe-pai, você pode adicionar qualquer atributo ou método novo necessários para diferenciar a classe-filha da classe-pai.

Vamos acrescentar um atributo que seja específico aos carros elétricos (uma bateria, por exemplo) e um método para mostrar esse atributo. Armazenaremos a capacidade da bateria e escreveremos um método que mostre uma descrição dela:

```
  class Car():
      --trecho omitido--

  class ElectricCar(Car):
      """Representa aspectos específicos de veículos elétricos."""

      def __init__(self, make, model, year):
          """
          Inicializa os atributos da classe pai
          Em seguida, inicializa os atributos específicos de um carro
  elétrico
          """
          super().__init__(make, model, year)
u         self.battery_size = 70

v     def describe_battery(self):
          """Exibe uma frase que descreve a capacidade da bateria."""
          print("This car has a " + str(self.battery_size) + "-kWh
  battery.")

  my_tesla = ElectricCar('tesla', 'model s', 2016)
  print(my_tesla.get_descriptive_name())
  my_tesla.describe_battery()
```

Em u adicionamos um novo atributo `self.battery_size` e definimos seu valor inicial, por exemplo, com `70`. Esse atributo será associado a todas as instâncias criadas a partir da classe `ElectricCar`, mas não será associado a nenhuma instância de `Car`. Também adicionamos um método chamado `describe_battery()`, que exibe informações sobre a bateria em v. Quando

chamamos esse método, temos uma descrição que é claramente específica de um carro elétrico:

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

Não há limites de quanto você pode especializar a classe `ElectricCar`. Você pode acrescentar a quantidade de atributos e métodos de que precisar para modelar um carro elétrico com qualquer grau de precisão necessário. Um atributo ou método que possa pertencer a qualquer carro, isto é, que não seja específico de um carro elétrico, deve ser adicionado à classe `Car`, em vez de ser colocado na classe `ElectricCar`. Então qualquer pessoa que usar a classe `Car` terá essa funcionalidade disponível também, e a classe `ElectricCar` conterá apenas códigos para as informações e comportamentos específicos de veículos elétricos.

Sobrescrevendo métodos da classe-pai

Qualquer método da classe-pai que não se enquadre no que você estiver tentando modelar com a classe-filha pode ser sobrescrito. Para isso, defina um método na classe-filha com o mesmo nome do método da classe-pai que você deseja sobrescrever. Python desprezará o método da classe-pai e só prestará atenção no método definido na classe-filha.

Suponha que a classe `Car` tenha um método chamado `fill_gas_tank()`. Esse método não faz sentido para um veículo totalmente elétrico, portanto você pode sobrescrever esse método. Eis uma maneira de fazer isso:

```
def ElectricCar(Car):
    --trecho omitido--

    def fill_gas_tank():
        """Carros elétricos não têm tanques de gasolina."""
        print("This car doesn't need a gas tank!")
```

Agora, se alguém tentar chamar `fill_gas_tank()` com um carro elétrico, Python ignorará esse método de `Car` e executará o código apresentado em seu lugar. Ao usar herança, você pode fazer suas classes-filhas preservarem o que for necessário e sobrescrever tudo o que não for utilizado da classe-

pai.

Instâncias como atributos

Ao modelar algo do mundo real no código você poderá perceber que está adicionando cada vez mais detalhes em uma classe. Poderá notar que há uma lista crescente de atributos e métodos e que seus arquivos estão começando a ficar extensos. Nessas situações, talvez você perceba que parte de uma classe pode ser escrita como uma classe separada. Sua classe maior poderá ser dividida em partes menores que funcionem em conjunto.

Por exemplo, se continuarmos adicionando detalhes à classe `ElectricCar`, podemos perceber que estamos acrescentando muitos atributos e métodos específicos à bateria do carro. Se percebermos que isso está acontecendo, podemos parar e transferir esses atributos e métodos para uma classe diferente chamada `Battery`. Então podemos usar uma instância de `Battery` como atributo da classe `ElectricCar`:

```
  class Car():
      --trecho omitido--

u class Battery():
      """Uma tentativa simples de modelar uma bateria para um carro
  elétrico."""

v     def __init__(self, battery_size=70):
          """Inicializa os atributos da bateria."""
          self.battery_size = battery_size

w     def describe_battery(self):
          """Exibe uma frase que descreve a capacidade da bateria."""
          print("This car has a " + str(self.battery_size) + "-kWh
  battery.")


  class ElectricCar(Car):
      """Representa aspectos específicos de veículos elétricos."""

      def __init__(self, make, model, year):
          """
```

```
        Inicializa os atributos da classe-pai
        Em seguida, inicializa os atributos específicos de um carro
elétrico
        """
        super().__init__(make, model, year)
x       self.battery = Battery()


my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

Em u definimos uma nova classe chamada `Battery` que não herda de nenhuma outra classe. O método `__init__()` em v tem um parâmetro, `battery_size`, além de `self`. É um parâmetro opcional que define a capacidade da bateria com 70 se nenhum valor for especificado. O método `describe_battery()` também foi transferido para essa classe w.

Na classe `ElectricCar`, adicionamos um atributo chamado `self.battery` x. Essa linha diz a Python para criar uma nova instância de `Battery` (com capacidade default de 70, pois não estamos especificando nenhum valor) e armazenar essa instância no atributo `self.battery`. Isso acontecerá sempre que o método `__init__()` for chamado; qualquer instância de `ElectricCar` agora terá uma instância de `Battery` criada automaticamente.

Criamos um carro elétrico e o armazenamos na variável `my_tesla`. Quando quisermos descrever a bateria, precisaremos trabalhar com o atributo `battery` do carro:

```
my_tesla.battery.describe_battery()
```

Essa linha diz a Python para usar a instância `my_tesla`, encontrar seu atributo `battery` e chamar o método `describe_battery()` associado à instância de `Battery` armazenada no atributo.

A saída é idêntica àquela que vimos antes:

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

Parece ser bastante trabalho extra, mas agora podemos descrever a

bateria com quantos detalhes quisermos sem deixar a classe `ElectricCar` entulhada. Vamos acrescentar outro método em `Battery` que informe a distância que o carro pode percorrer de acordo com a capacidade da bateria:

```
  class EuroCalc
      --trecho omitido--

  class Battery():
      --trecho omitido--

u     def get_range(self):
          """Exibe uma frase sobre a distância que o carro é capaz de
  percorrer com essa bateria."""
         if self.battery_size == 70:
              range = 240
          elif self.battery_size == 85:
              range = 270

          message = "This car can go approximately " + str(range)
          message += " miles on a full charge."
          print(message)

  class ElectricCar(Car):
      --trecho omitido--

  my_tesla = ElectricCar('tesla', 'model s', 2016)
  print(my_tesla.get_descriptive_name())
  my_tesla.battery.describe_battery()
v my_tesla.battery.get_range()
```

O novo método `get_range()` em **u** efetua uma análise simples. Se a capacidade da bateria for de 70 kWh, `get_range()` define o alcance do carro com 240 milhas; se a capacidade for de 85 kWh, o alcance será definido com 270 milhas. Esse valor é então apresentado. Quando quisermos usar esse método, novamente, devemos chamá-lo por meio do atributo `battery` do carro **v**.

A saída nos informa a distância que o carro é capaz de percorrer de acordo com a capacidade de sua bateria.

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

Modelando objetos do mundo real

À medida que começar a modelar itens mais complexos como carros elétricos, você vai ter que encarar perguntas interessantes. A distância que um carro elétrico é capaz de percorrer é uma propriedade da bateria ou do carro? Se estivermos descrevendo apenas um carro, provavelmente não haverá problemas em manter a associação do método `get_range()` com a classe `Battery`. Entretanto, se estivermos descrevendo toda uma linha de carros de um fabricante, é provável que vamos querer transferir `get_range()` para a classe `ElectricCar`. O método `get_range()` continuaria verificando a capacidade da bateria antes de determinar a distância que o carro é capaz de percorrer, mas informaria um alcance específico para o tipo de carro com o qual está associado. De modo alternativo, poderíamos manter a associação entre o método `get_range()` e a bateria, mas passaríamos um parâmetro a ele, por exemplo, `car_model`. O método `get_range()` então informaria a distância que o carro poderá percorrer de acordo com a capacidade da bateria e o modelo do carro.

Isso leva você a um ponto interessante em seu crescimento como programador. Quando tiver que encarar questões como essa, você estará pensando em um nível lógico mais alto, em vez de se concentrar no nível da sintaxe. Estará pensando não em Python, mas no modo de representar o mundo real como um código. Quando atingir esse ponto, você perceberá que, muitas vezes, não há abordagens certas ou erradas para modelar situações do mundo real. Algumas abordagens são mais eficientes que outras, mas descobrir as representações mais eficientes exige prática. Se seu código estiver funcionando conforme desejado, é sinal de que você está se saindo bem! Não desanime se perceber que está destruindo suas classes e reescrevendo-as várias vezes usando diferentes abordagens. No caminho para escrever um código preciso e eficiente, todos passam por esse processo.

FAÇA VOCÊ MESMO

9.6 – Sorveteria: Uma sorveteria é um tipo específico de restaurante. Escreva uma classe chamada `IceCreamStand` que herde da classe `Restaurant` escrita no Exercício 9.1 (página 225) ou no Exercício 9.4 (página 232). Qualquer versão da classe funcionará; basta escolher aquela de que você mais gosta. Adicione um atributo chamado `flavors` que armazene uma lista de sabores de sorvete. Escreva um método para mostrar esses sabores. Crie uma instância de `IceCreamStand` e chame esse método.

9.7 – Admin: Um administrador é um tipo especial de usuário. Escreva uma classe chamada `Admin` que herde da classe `User` escrita no Exercício 9.3 (página 226), ou no Exercício 9.5 (página 232). Adicione um atributo `privileges` que armazene uma lista de strings como `"can add post"`, `"can delete post"` `"can ban user"`, e assim por diante. Escreva um método chamado `show_privileges()` que liste o conjunto de privilégios de um administrador. Crie uma instância de `Admin` e chame seu método.

9.8 – Privilégios: Escreva uma classe `Privileges` separada. A classe deve ter um atributo `privileges` que armazene uma lista de strings conforme descrita no Exercício 9.7. Transfira o método `show_privileges()` para essa classe. Crie uma instância de `Privileges` como um atributo da classe `Admin`. Crie uma nova instância de `Admin` e use seu método para exibir os privilégios.

9.9 – Upgrade de bateria: Use a última versão de *electric_car.py* desta seção. Acrescente um método chamado `upgrade_battery()` na classe `Battery`. Esse método deve verificar a capacidade da bateria e defini-la com 85 se o valor for diferente. Crie um carro elétrico com uma capacidade de bateria default, chame `get_range()` uma vez e, em seguida, chame `get_range()` uma segunda vez após fazer um upgrade da bateria. Você deverá ver um aumento na distância que o carro é capaz de percorrer.

Importando classes

À medida que acrescentar mais funcionalidades às classes, seus arquivos ficarão maiores, mesmo quando usar herança de forma apropriada. Para estar de acordo com a filosofia geral de Python, quanto menos entulhados estiverem seus arquivos, melhor será. Para ajudar, Python permite armazenar classes em módulos e então importar as classes necessárias em seu programa principal.

Importando uma única classe

Vamos criar um módulo que contenha apenas a classe `Car`. Isso cria um problema sutil de nomenclatura: já temos um arquivo chamado *car.py* neste capítulo, mas esse módulo deve se chamar *car.py* porque contém código que representa um carro. Resolveremos esse problema de nomenclatura armazenando a classe `Car` em um módulo chamado *car.py*, substituindo o arquivo *car.py* que estávamos usando antes. A partir de agora, qualquer programa que use esse módulo precisará de um nome de arquivo mais específico, por exemplo, *my_car.py*. Eis o arquivo *car.py* somente com o código da classe `Car`:

car.py

```
u """Uma classe que pode ser usada para representar um carro."""

  class Car():
      """Uma tentativa simples de representar um carro."""

      def __init__(self, make, model, year):
          """Inicializa os atributos que descrevem um carro."""
          self.make = make
          self.model = model
          self.year = year
          self.odometer_reading = 0

      def get_descriptive_name(self):
          """Devolve um nome descritivo formatado de modo elegante."""
          long_name = str(self.year) + ' ' + self.make + ' ' + self.model
          return long_name.title()

      def read_odometer(self):
          """Exibe uma frase que mostra a milhagem do carro."""
          print("This car has " + str(self.odometer_reading) + " miles on
  it.")

      def update_odometer(self, mileage):
          """
          Define o valor de leitura do hodômetro com o valor especificado
          Rejeita alteração se for tentativa de definir um valor menor
  para o hodômetro
          """
```

```
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do
hodômetro."""
        self.odometer_reading += miles
```

Em u incluímos uma docstring no nível de módulo que descreve rapidamente o conteúdo desse módulo. Escreva uma docstring para cada módulo que criar.

Agora criamos um arquivo separado chamado *my_car.py*. Esse arquivo importará a classe `Car` e então criará uma instância dessa classe:

my_car.py

```
u from car import Car

  my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car.get_descriptive_name())

  my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

A instrução `import` em u diz a Python para abrir o módulo `car` e importar a classe `Car`. Agora podemos usar a classe `Car` como se ela estivesse definida nesse arquivo. A saída é a mesma que vimos antes:

```
2016 Audi A4
This car has 23 miles on it.
```

Importar classes é uma maneira eficiente de programar. Imagine o tamanho que esse programa teria se a classe `Car` inteira estivesse incluída. Quando transferimos a classe para um módulo e o importamos, continuamos usufruindo da mesma funcionalidade, porém o arquivo com o programa principal permanece limpo e fácil de ler. Também armazenamos a maior parte da lógica em arquivos separados; depois que suas classes estiverem funcionando conforme esperado, você poderá deixar

de lado esses arquivos e se concentrar na lógica de mais alto nível de seu programa principal.

Armazenando várias classes em um módulo

Você pode armazenar tantas classes quantas forem necessárias em um único módulo, embora cada classe em um módulo deva estar, de algum modo, relacionada com outra classe. Ambas as classes, `Battery` e `ElectricCar`, ajudam a representar carros, portanto vamos acrescentá-las ao módulo *car.py*:

car.py

```
"""Um conjunto de classes usado para representar carros à gasolina e
elétricos."""

class Car():
    --trecho omitido--

class Battery():
    """Uma tentativa simples de modelar uma bateria para um carro
elétrico."""

    def __init__(self, battery_size=60):
        """Inicializa os atributos da bateria."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Exibe uma frase que descreve a capacidade da bateria."""
        print("This car has a " + str(self.battery_size) + "-kWh
battery.")

    def get_range(self):
        """Exibe frase sobre a distância que o carro pode percorrer com
essa bateria."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
```

```
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):
    """Modela aspectos de um carro específicos de veículos
elétricos."""

    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos de um carro
elétrico.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Agora podemos criar um novo arquivo chamado *my_electric_car.py*, importar a classe `ElectricCar` e criar um carro elétrico:

my_electric_car.py

```
from car import ElectricCar
my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Esse código gera a mesma saída que vimos antes, embora a maior parte da lógica esteja oculta em um módulo:

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

Importando várias classes de um módulo

Podemos importar quantas classes forem necessárias em um arquivo de programa. Se quisermos criar um carro comum e um carro elétrico no mesmo arquivo, precisaremos importar tanto a classe `Car` quanto a classe `ElectricCar`:

my_cars.py

```
u from car import Car, ElectricCar

v my_beetle = Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

w my_tesla = ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

Importe várias classes de um módulo separando cada classe com uma vírgula u. Depois que importar as classes de que precisará, você poderá criar quantas instâncias de cada classe quantas forem necessárias.

Nesse exemplo, criamos um Volkswagen Beetle comum em v e um Tesla Roadster elétrico em w:

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

Importando um módulo completo

Também podemos importar um módulo completo e então acessar as classes necessárias usando a notação de ponto. Essa abordagem é simples e resulta em um código fácil de ler. Como toda chamada que cria uma instância de uma classe inclui o nome do módulo, você não terá conflito de nomes com qualquer nome usado no arquivo atual.

Eis a aparência do código para importar o módulo `car` inteiro e então criar um carro comum e um carro elétrico:

my_cars.py

```
u import car

v my_beetle = car.Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

w my_tesla = car.ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

Em u importamos o módulo `car` inteiro. Então acessamos as classes

necessárias por meio da sintaxe *nome_do_módulo.nome_da_classe*. Em v criamos novamente um Volkswagen Beetle e em w, um Tesla Roadster.

Importando todas as classes de um módulo

Você pode importar todas as classes de um módulo usando a sintaxe a seguir:

```
from nome_do_módulo import *
```

Esse método não é recomendado por dois motivos. Em primeiro lugar, é conveniente ser capaz de ler as instruções `import` no início de um arquivo e ter uma noção clara de quais classes um programa utiliza. Com essa abordagem, não fica claro quais são as classes do módulo que você está usando. Essa abordagem também pode resultar em confusão com nomes presentes no arquivo. Se você acidentalmente importar uma classe com o mesmo nome que outro item em seu arquivo de programa, poderá gerar erros difíceis de serem diagnosticados. Estou mostrando essa opção aqui porque, embora não seja uma abordagem recomendada, é provável que você vá vê-la no código de outras pessoas.

Se precisar importar muitas classes de um módulo, é melhor importar o módulo todo e usar a sintaxe *nome_do_módulo.nome_da_classe*. Você não verá todas as classes usadas no início do arquivo, mas verá claramente em que lugares o módulo é utilizado no programa. Possíveis conflitos de nomes que possam ocorrer ao importar todas as classes de um módulo também serão evitados.

Importando um módulo em um módulo

Às vezes, você vai querer espalhar suas classes em vários módulos para impedir que um arquivo cresça demais e evitar a armazenagem de classes não relacionadas no mesmo módulo. Ao armazenar suas classes em vários módulos, você poderá descobrir que uma classe em um módulo depende de uma classe que está em outro módulo. Se isso acontecer, importe a classe necessária no primeiro módulo.

Por exemplo, vamos armazenar a classe `Car` em um módulo e as classes

ElectricCar e Battery em um módulo separado. Criaremos um novo módulo chamado *electric_car.py* – substituindo o arquivo *electric_car.py* que criamos antes – e copiaremos apenas as classes Battery e ElectricCar para esse arquivo:

electric_car.py

```
  """Um conjunto de classes que pode ser usado para representar carros
  elétricos."""

u from car import Car

  class Battery():
      --trecho omitido--

  class ElectricCar(Car):
      --trecho omitido--
```

A classe ElectricCar precisa ter acesso à sua classe-pai Car, portanto importamos Car diretamente para o módulo em u. Se esquecermos de colocar essa linha, Python gerará um erro quando tentarmos criar uma instância de ElectricCar. Também precisamos atualizar o módulo Car para que ele contenha apenas a classe Car:

car.py

```
  """Uma classe que pode ser usada para representar um carro."""

  class Car():
      --trecho omitido--
```

Agora podemos fazer a importação de cada módulo separadamente e criar o tipo de carro que for necessário:

my_cars.py

```
u from car import Car
  from electric_car import ElectricCar

  my_beetle = Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())
```

```
my_tesla = ElectricCar('tesla', 'roadster', 2016)
print(my_tesla.get_descriptive_name())
```

Em u importamos `Car` de seu módulo e `ElectricCar` de seu módulo. Em seguida, criamos um carro comum e um carro elétrico. Os dois tipos de carro são criados corretamente:

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

Definindo o seu próprio fluxo de trabalho

Como podemos ver, Python oferece muitas opções para estruturar o código em um projeto de grande porte. É importante conhecer todas essas possibilidades para que você possa determinar a melhor maneira de organizar seus projetos, assim como entender o projeto de outras pessoas.

Quando estiver começando a programar, mantenha a estrutura de seu código simples. Procure fazer tudo em um só arquivo e transfira suas classes para módulos separados depois que tudo estiver funcionando. Se gostar do modo como os módulos e os arquivos interagem, experimente armazenar suas classes em módulos quando iniciar um projeto. Encontre uma abordagem que permita escrever um código que funcione, e comece a partir daí.

FAÇA VOCÊ MESMO

9.10 – Importando Restaurant: Usando sua classe `Restaurant` mais recente, armazene-a em um módulo. Crie um arquivo separado que importe `Restaurant`. Crie uma instância de `Restaurant` e chame um de seus métodos para mostrar que a instrução `import` funciona de forma apropriada.

9.11 – Importando Admin: Comece com seu programa do Exercício 9.8 (página 241). Armazene as classes `User`, `Privileges` e `Admin` em um módulo. Crie um arquivo separado e uma instância de `Admin` e chame `show_privileges()` para mostrar que tudo está funcionando de forma apropriada.

9.12 – Vários módulos: Armazene a classe `User` em um módulo e as classes `Privileges` e `Admin` em um módulo separado. Em outro arquivo, crie uma instância de `Admin` e chame `show_privileges()` para mostrar que tudo continua funcionando de forma apropriada.

Biblioteca-padrão de Python

A *biblioteca-padrão de Python* é um conjunto de módulos incluído em todas as instalações de Python. Agora que temos uma compreensão básica de como as classes funcionam, podemos começar a usar módulos como esses, escritos por outros programadores. Podemos usar qualquer função ou classe da biblioteca-padrão incluindo uma instrução `import` simples no início do arquivo. Vamos analisar a classe `OrderedDict` do módulo `collections`.

Os dicionários permitem associar informações, mas eles não mantêm um controle da ordem em que os pares chave-valor são acrescentados. Se você estiver criando um dicionário e quiser manter o controle da ordem em que os pares chave-valor são adicionados, a classe `OrderedDict` do módulo `collections` poderá ser usada. Instâncias da classe `OrderedDict` se comportam quase do mesmo modo que os dicionários, exceto que mantêm o controle da ordem em que os pares chave-valor são adicionados.

Vamos retomar o exemplo *favorite_languages.py* do Capítulo 6. Dessa vez, vamos manter o controle da ordem em que as pessoas responderam à enquete:

favorite_languages.py

```
u from collections import OrderedDict

v favorite_languages = OrderedDict()

w favorite_languages['jen'] = 'python'
  favorite_languages['sarah'] = 'c'
  favorite_languages['edward'] = 'ruby'
  favorite_languages['phil'] = 'python'

x for name, language in favorite_languages.items():
      print(name.title() + "'s favorite language is " +
          language.title() + ".")
```

Começamos importando a classe `OrderedDict` do módulo `collections` em u. Em v criamos uma instância da classe `OrderedDict` e a armazenamos em `favorite_languages`. Observe que não há chaves; a chamada a `OrderedDict()`

cria um dicionário ordenado vazio para nós e o armazena em `favorite_languages`. Então adicionamos cada nome e linguagem em `favorite_languages`, um de cada vez **W**. Agora, quando percorremos `favorite_languages` com um laço em **X**, sabemos que sempre teremos as respostas na ordem em que os itens foram adicionados:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

É uma ótima classe para conhecer, pois combina a principal vantagem das listas (preservar a ordem original) com o principal recurso dos dicionários (associar informações). Quando começar a modelar situações do mundo real que sejam de seu interesse, é provável que você vá se deparar com uma situação em que um dicionário ordenado seja exatamente o que você precisa. À medida que conhecer melhor a biblioteca-padrão, você passará a ter familiaridade com vários módulos como esse, que ajudam a tratar situações comuns.

NOTA Você também pode fazer download de módulos de fontes externas. Veremos vários desses exemplos na Parte II, quando precisaremos de módulos externos para concluir cada projeto.

FAÇA VOCÊ MESMO

9.13 – Reescrevendo o programa com OrderedDict: Comece com o Exercício 6.4 (página 155), em que usamos um dicionário-padrão para representar um glossário. Reescreva o programa usando a classe `OrderedDict` e certifique-se de que a ordem da saída coincida com a ordem em que os pares chave-valor foram adicionados ao dicionário.

9.14 – Dados: O módulo `random` contém funções que geram números aleatórios de várias maneiras. A função `randint()` devolve um inteiro no intervalo especificado por você. O código a seguir devolve um número entre 1 e 6:

```
from random import randint
x = randint(1, 6)
```

Crie uma classe `Die` com um atributo chamado `sides`, cujo valor default é 6. Escreva um método chamado `roll_die()` que exiba um número aleatório entre 1 e

o número de lados do dado. Crie um dado de seis dados e lance-o dez vezes.

Crie um dado de dez lados e outro de vinte lados. Lance cada dado dez vezes.

9.15 – Módulo Python da semana: Um excelente recurso para explorar a biblioteca-padrão de Python é um site chamado *Python Module of the Week* (Módulo Python da semana). Acesse *http://pymotw.com/* e observe a tabela de conteúdo. Encontre um módulo que pareça ser interessante e leia a sua descrição ou explore a documentação dos módulos `collections` e `random`.

Estilizando classes

Vale a pena esclarecer algumas questões ligadas à estilização de classes, em especial quando seus programas se tornarem mais complexos.

Os nomes das classes devem ser escritos com *CamelCaps*. Para isso, cada palavra do nome deve ter a primeira letra maiúscula, e você não deve usar underscores. Nomes de instâncias e de módulos devem ser escritos com letras minúsculas e underscores entre as palavras.

Toda classe deve ter uma docstring logo depois de sua definição. A docstring deve conter uma breve descrição do que a classe faz, e as mesmas convenções de formatação devem ser usadas para escrever docstrings em funções. Cada módulo também deve ter uma docstring que descreva para que servem as classes de um módulo.

Podemos usar linhas em branco para organizar o código, mas você não deve utilizá-las de modo excessivo. Em uma classe, podemos usar uma linha em branco entre os métodos; em um módulo, podemos usar duas linhas em branco para separar as classes.

Se houver necessidade de importar um módulo da biblioteca-padrão e um módulo escrito por você, coloque a instrução de importação do módulo da biblioteca-padrão antes. Então acrescente uma linha em branco e a instrução de importação para o módulo que você escreveu. Em programas com várias instruções de importação, essa convenção facilita ver a origem dos diferentes módulos utilizados pelo programa.

Resumo

Neste capítulo, vimos como escrever nossas próprias classes. Aprendemos

a armazenar informações em uma classe usando atributos e a escrever métodos que conferem o comportamento necessário às suas classes. Vimos como escrever métodos `__init__()` que criam instâncias de suas classes com os atributos específicos que desejamos ter. Aprendemos a modificar os atributos de uma instância diretamente e por meio de métodos. Vimos que a herança pode simplificar a criação de classes relacionadas umas às outras e aprendemos a usar instâncias de uma classe como atributos de outra para deixá-las mais simples.

Aprendemos a armazenar classes em módulos e a importar as classes necessárias nos arquivos em que elas serão usadas para deixar seus projetos organizados. Começamos a conhecer a biblioteca-padrão de Python e vimos um exemplo com a classe `OrderedDict` do módulo `collections`. Por fim, vimos como estilizar classes usando as convenções de Python.

No Capítulo 10, aprenderemos a trabalhar com arquivos para que possamos salvar o trabalho feito por você e pelos usuários de seu programa. Também conheceremos as *exceções*: uma classe Python especial, concebida para ajudar você a responder a erros quando esses surgirem.

10
ARQUIVOS E EXCEÇÕES

Agora que você já dominou as habilidades básicas necessárias para escrever programas organizados, fáceis de usar, é hora de pensar em deixar seus programas mais relevantes e utilizáveis. Neste capítulo aprenderemos a trabalhar com arquivos para que seus programas possam analisar rapidamente muitos dados.

Veremos como tratar erros a fim de que seus programas não falhem quando se depararem com situações inesperadas. Conheceremos as *exceções* – objetos Python especiais para administrar erros que surgirem enquanto um programa estiver executando. Também conheceremos o módulo `json`, que permite salvar dados de usuário para que não sejam perdidos quando seu programa parar de executar.

Aprender a trabalhar com arquivos e a salvar dados deixará seus programas mais fáceis de usar. Os usuários poderão escolher quais dados devem fornecer e quando. As pessoas podem executar seu programa, fazer alguma tarefa e então fechá-lo e retomá-lo mais tarde, do ponto em que pararam. Aprender a tratar exceções ajudará você a lidar com situações em que os arquivos não existam e com outros problemas que possam fazer seus programas falharem. Isso deixará seus programas mais robustos quando dados ruins forem encontrados, sejam eles provenientes de erros inocentes ou de tentativas maliciosas de fazer seus programas falharem. Com as habilidades desenvolvidas neste capítulo, você deixará seus programas mais aplicáveis, utilizáveis e estáveis.

Lendo dados de um arquivo

Uma quantidade incrível de dados está disponível em arquivos-texto. Os arquivos-texto podem conter dados meteorológicos, de tráfego, socioeconômicos, trabalhos literários e outros. Ler dados de um arquivo é particularmente útil em aplicações de análise de dados, mas também se aplica a qualquer situação em que você queira analisar ou modificar

informações armazenadas em um arquivo. Por exemplo, podemos escrever um programa que leia o conteúdo de um arquivo-texto e reescreva o arquivo com uma formatação que permita a um navegador exibi-lo.

Quando quiser trabalhar com as informações de um arquivo-texto, o primeiro passo será ler o arquivo em memória. Você pode ler todo o conteúdo de um arquivo ou pode trabalhar com uma linha de cada vez.

Lendo um arquivo inteiro

Para começar, precisamos de um arquivo com algumas linhas de texto. Vamos iniciar com um arquivo que contenha o valor de *pi* com trinta casas decimais, dez casas por linha:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

Para testar esses exemplos por conta própria, você pode inserir essas linhas em um editor e salvar o arquivo como *pi_digits.txt*, ou pode fazer o download do arquivo a partir da página de recursos do livro em *https://www.nostarch.com/pythoncrashcourse/*. Salve o arquivo no mesmo diretório em que você armazenará os programas deste capítulo.

Aqui está um programa que abre esse arquivo, lê seus dados e exibe o conteúdo na tela:

file_reader.py

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents)
```

Muitas atividades acontecem na primeira linha desse programa. Vamos começar observando a função `open()`. Para realizar qualquer tarefa com um arquivo, mesmo que seja apenas exibir o seu conteúdo, você precisará inicialmente *abrir* o arquivo para acessá-lo. A função `open()` precisa de um argumento: o nome do arquivo que você quer abrir. Python procura esse arquivo no diretório em que o programa executando no momento está

armazenado. Nesse exemplo, *file_reader.py* está executando, portanto Python procura *pi_digits.txt* no diretório em que o arquivo *file_reader.py* está armazenado. A função `open()` devolve um objeto que representa o arquivo. Nesse caso, `open('pi_digits.txt')` devolve um objeto que representa *pi_digits.txt*. Python armazena esse objeto em `file_object`, com o qual trabalharemos posteriormente no programa.

A palavra reservada `with` fecha o arquivo depois que não for mais necessário acessá-lo. Observe como chamamos `open()` nesse programa, mas não chamamos `close()`. Você poderia abrir e fechar o arquivo chamando `open()` e `close()`, mas se um bug em seu programa impedir que a instrução `close()` seja executada, o arquivo não será fechado. Isso pode parecer trivial, mas arquivos indevidamente fechados podem provocar perda de dados ou estes podem ser corrompidos. Além disso, se `close()` for chamado cedo demais em seu programa, você se verá tentando trabalhar com um arquivo *fechado* (um arquivo que não pode ser acessado), o que resultará em mais erros. Nem sempre é fácil saber exatamente quando devemos fechar um arquivo, mas com a estrutura mostrada aqui, Python descobrirá isso para você. Tudo que você precisa fazer é abrir o arquivo e trabalhar com ele conforme desejado, com a confiança de que Python o fechará automaticamente no momento certo.

Depois que tivermos um objeto arquivo que represente *pi_digits.txt*, usamos o método `read()` na segunda linha de nosso programa para ler todo o conteúdo do arquivo e armazená-lo em uma longa string em `contents`. Quando exibimos o valor de `contents`, vemos o arquivo-texto completo:

```
3.1415926535
  8979323846
  2643383279
```

A única diferença entre essa saída e o arquivo original é a linha em branco extra no final da saída. A linha em branco aparece porque `read()` devolve uma string vazia quando alcança o final do arquivo; essa string vazia aparece como uma linha em branco. Se quiser remover essa linha em branco extra, `rstrip()` pode ser usada na instrução `print`:

```
with open('pi_digits.txt') as file_object:
```

```
    contents = file_object.read()
    print(contents.rstrip())
```

Lembre-se de que o método `rstrip()` de Python remove qualquer caractere branco do lado direito de uma string. Agora a saída será exatamente igual ao conteúdo do arquivo original:

```
3.1415926535
  8979323846
  2643383279
```

Paths de arquivo

Quando um nome de arquivo simples como *pi_digits.txt* é passado para a função `open()`, Python observa o diretório em que o arquivo executado no momento (isto é, seu arquivo de programa *.py*) está armazenado.

Às vezes, dependendo de como o seu trabalho estiver organizado, o arquivo que você quer abrir não estará no mesmo diretório que o seu arquivo de programa. Por exemplo, você pode armazenar seus arquivos de programa em uma pasta chamada *python_work*; nessa pasta pode haver outra pasta chamada *text_files* para distinguir seus arquivos de programa dos arquivos-texto que eles manipulam. Apesar de *text_files* estar em *python_work*, simplesmente passar o nome de um arquivo que está em *text_files* para `open()` não funcionará, pois Python procurará o arquivo apenas em *python_work*; ele não prosseguirá procurando em *text_files*. Para fazer Python abrir arquivos de um diretório que não seja aquele em que seu arquivo de programa está armazenado, é preciso fornecer um *path de arquivo*, que diz a Python para procurar em um local específico de seu sistema.

Como *text_files* está em *python_work*, você pode usar um path de arquivo relativo para abrir um arquivo em *text_files*. Um *path de arquivo relativo* diz a Python para procurar um local especificado, relativo ao diretório em que o arquivo de programa em execução no momento está armazenado. No Linux e no OS X, você escreveria o seguinte:

```
with open('text_files/nome_do_arquivo.txt') as file_object:
```

Essa linha diz a Python para procurar o arquivo *.txt* desejado na pasta

text_files e supõe que essa pasta está localizada em *python_work* (e está). Em sistemas Windows, use uma barra invertida (\) no lugar da barra para a frente (/) no path do arquivo:

```
with open('text_files\nome_do_arquivo.txt') as file_object:
```

Você também pode dizer a Python exatamente em que local está o arquivo em seu computador, não importando o lugar em que o programa em execução no momento esteja armazenado. Isso é chamado de *path absoluto do arquivo*. Utilize um path absoluto se um path relativo não funcionar. Por exemplo, se você colocou *text_files* em alguma pasta diferente de *python_work* – por exemplo, em uma pasta chamada *other_files* – então simplesmente passar o path `'text_files/nome_do_arquivo.txt'` para `open()` não funcionará porque Python procurará esse local somente em *python_work*. Você precisará fornecer um path completo para deixar claro em que lugar você quer que Python procure.

Paths absolutos geralmente são mais longos que paths relativos, portanto é conveniente armazená-los em uma variável e então passar essa variável para `open()`. No Linux e no OS X, paths absolutos têm o seguinte aspecto:

```
file_path =
'/home/ehmatthes/other_files/text_files/nome_do_arquivo.txt'
with open(file_path) as file_object:
```

No Windows, eles se parecem com:

```
file_path =
'C:\Users\ehmatthes\other_files\text_files\nome_do_arquivo.txt'
with open(file_path) as file_object:
```

Ao usar paths absolutos, podemos ler arquivos de qualquer lugar do sistema. Por enquanto, é mais fácil armazenar arquivos no mesmo diretório em que estão seus arquivos de programa ou em uma pasta como *text_files* no diretório em que estiverem seus arquivos de programa.

NOTA Sistemas Windows às vezes interpretam corretamente as barras para a frente nos paths de arquivo. Se você usa Windows e não está obtendo os resultados esperados, tente usar barras invertidas.

Lendo dados linha a linha

Quando estiver lendo um arquivo, com frequência você vai querer analisar cada linha do arquivo. Talvez você esteja procurando determinada informação no arquivo ou queira modificar o texto do arquivo de alguma maneira. Por exemplo, você pode ler um arquivo de dados meteorológicos e trabalhar com qualquer linha que inclua a palavra *ensolarado* na descrição da previsão do tempo para esse dia. Em uma notícia, talvez você queira procurar todas as linhas com a tag `<headline>` e reescrever essa linha com um tipo específico de formatação.

Podemos usar um laço `for` no objeto arquivo para analisar cada uma de suas linhas, uma de cada vez:

file_reader.py

```
u filename = 'pi_digits.txt'

v with open(filename) as file_object:
w     for line in file_object:
          print(line)
```

Em u armazenamos o nome do arquivo que estamos lendo em uma variável `filename`. Essa é uma convenção comum quando trabalhamos com arquivos. Como a variável `filename` não representa o arquivo propriamente dito – é apenas uma string que diz a Python em que lugar o arquivo se encontra – você pode facilmente trocar `'pi_digits.txt'` pelo nome de outro arquivo com o qual você queira trabalhar. Depois da chamada a `open()`, um objeto que representa o arquivo e seu conteúdo é armazenado na variável `file_object` v. Novamente, usamos a sintaxe `with` para deixar Python abrir e fechar o arquivo de modo apropriado. Para analisar o conteúdo do arquivo, trabalhamos com cada linha do arquivo percorrendo o objeto arquivo em um laço w.

Quando exibimos cada linha, encontramos outras linhas em branco:

```
3.1415926535

  8979323846
```

```
  2643383279
```

Essas linhas em branco aparecem porque um caractere invisível de quebra de linha está no final de cada linha do arquivo-texto. A instrução `print` adiciona a sua própria quebra de linha sempre que a chamamos, portanto acabamos com dois caracteres de quebra de linha no final de cada linha: um do arquivo e outro da instrução `print`. Se usarmos `rstrip()` em cada linha na instrução `print`, eliminamos essas linhas em branco extras:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Agora a saída é igual ao conteúdo do arquivo novamente:

```
3.1415926535
  8979323846
  2643383279
```

Criando uma lista de linhas de um arquivo

Quando usamos `with`, o objeto arquivo devolvido por `open()` estará disponível somente no bloco `with` que o contém. Se quiser preservar o acesso ao conteúdo de um arquivo fora do bloco `with`, você pode armazenar as linhas do arquivo em uma lista dentro do bloco e então trabalhar com essa lista. Pode processar partes do arquivo imediatamente e postergar parte do processamento de modo que seja feito mais tarde no programa.

O exemplo a seguir armazena as linhas de *pi_digits.txt* em uma lista no bloco `with` e, em seguida, exibe as linhas fora desse bloco:

```
  filename = 'pi_digits.txt'

  with open(filename) as file_object:
u     lines = file_object.readlines()

v for line in lines:
      print(line.rstrip())
```

Em u o método readlines() armazena cada linha do arquivo em uma lista. Essa lista é então armazenada em lines, com a qual podemos continuar trabalhando depois que o bloco with terminar. Em v usamos um laço for simples para exibir cada linha de lines. Como cada item de lines corresponde a uma linha do arquivo, a saída será exatamente igual ao conteúdo do arquivo.

Trabalhando com o conteúdo de um arquivo

Depois de ler um arquivo em memória, você poderá fazer o que quiser com esses dados; desse modo, vamos explorar rapidamente os dígitos de *pi*. Em primeiro lugar, tentaremos criar uma única string contendo todos os dígitos do arquivo, sem espaços em branco:

pi_string.py

```
  filename = 'pi_digits.txt'

  with open(filename) as file_object:
      lines = file_object.readlines()

u pi_string = ''
v for line in lines:
      pi_string += line.rstrip()

w print(pi_string)
  print(len(pi_string))
```

Começamos abrindo o arquivo e armazenando cada linha de dígitos em uma lista, como fizemos no exemplo anterior. Em u criamos uma variável pi_string para armazenar os dígitos de *pi*. Então criamos um laço que acrescenta cada uma das linhas de dígitos em pi_string removendo o caractere de quebra de linha v. Em w exibimos essa string e mostramos também o seu tamanho:

```
3.1415926535  8979323846  2643383279
36
```

A variável pi_string contém os espaços em branco que estavam do lado esquerdo dos dígitos em cada linha, mas podemos nos livrar deles usando

`strip()` no lugar de `rstrip()`:

```
filename = 'pi_30_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

Agora temos uma string que contém *pi* com trinta casas decimais. A string tem 32 caracteres de tamanho porque inclui também o número 3 na frente e um ponto decimal:

```
3.141592653589793238462643383279
32
```

NOTA Quando Python lê um arquivo-texto, todo o texto do arquivo é interpretado como uma string. Se você ler um número e quiser trabalhar com esse valor em um contexto numérico, será necessário convertê-lo em um inteiro usando a função `int()` ou convertê-lo em um número de ponto flutuante com a função `float()`.

Arquivos grandes: um milhão de dígitos

Até agora, nosso enfoque foi analisar um arquivo-texto que continha apenas três linhas, mas o código desses exemplos também funcionará bem em arquivos muito maiores. Se começarmos com um arquivo-texto que contenha *pi* com um milhão de casas decimais, e não trinta, uma única string contendo todos esses dígitos poderá ser criada. Não precisamos alterar nada em nosso programa, exceto para lhe passar um arquivo diferente. Além disso, exibiremos as cinquenta primeiras casas decimais para que não seja necessário assistirmos a um milhão de dígitos rolando pelo terminal:

pi_string.py

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.strip()

print(pi_string[:52] + "...")
print(len(pi_string))
```

A saída mostra que, realmente, temos uma string contendo *pi* com um milhão de casas decimais:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python não tem nenhum limite inerente para a quantidade de dados com que podemos trabalhar; podemos trabalhar com tantos dados quantos a memória de seu sistema for capaz de tratar.

NOTA Para executar esse programa (e vários dos próximos exemplos), você deverá fazer download dos recursos disponíveis em *https://www.nostarch.com/pythoncrashcourse/*.

Seu aniversário está contido em pi?

Sempre tive curiosidade de saber se a minha data de nascimento aparece em algum lugar nos dígitos de *pi*. Vamos usar o programa que acabamos de escrever para descobrir se a data de nascimento de alguém aparece em algum ponto no primeiro milhão de dígitos de *pi*. Podemos fazer isso expressando cada data de nascimento como uma string de dígitos e verificando se essa string aparece em algum ponto de `pi_string`:

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
```

```
for line in lines:
    pi_string += line.rstrip()

u birthday = input("Enter your birthday, in the form mmddyy: ")
v if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of
pi.")
```

Em u pedimos a data de nascimento do usuário e, em seguida, em v, verificamos se essa string está em `pi_string`. Vamos testar isso:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

Minha data de nascimento está nos dígitos de *pi*! Depois de ter lido um arquivo, podemos analisar seu conteúdo de praticamente qualquer modo que pudermos imaginar.

FAÇA VOCÊ MESMO

10.1 – Aprendendo Python: Abra um arquivo em branco em seu editor de texto e escreva algumas linhas que sintetizem o que você aprendeu sobre Python até agora. Comece cada linha com a expressão *Em Python podemos...*. Salve o arquivo como *learning_python.txt* no mesmo diretório em que estão seus exercícios deste capítulo. Escreva um programa que leia o arquivo e mostre o que você escreveu, três vezes. Exiba o conteúdo uma vez lendo o arquivo todo, uma vez percorrendo o objeto arquivo com um laço e outra armazenando as linhas em uma lista e então trabalhando com ela fora do bloco `with`.

10.2 – Aprendendo C: Você pode usar o método `replace()` para substituir qualquer palavra por uma palavra diferente em uma string. Eis um exemplo rápido que mostra como substituir a palavra `'dog'` por `'cat'` em uma frase:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Leia cada linha do arquivo *learning_python.txt* que você acabou de criar e substitua a palavra *Python* pelo nome de outra linguagem, por exemplo, C. Mostre cada linha modificada na tela.

Escrevendo dados em um arquivo

Uma das maneiras mais simples de salvar dados é escrevê-los em um arquivo. Quando um texto é escrito em um arquivo, o resultado estará disponível depois que você fechar o terminal que contém a saída de seu programa. Podemos analisar a saída depois que um programa acabar de executar e compartilhar os arquivos de saída com outras pessoas também. Além disso, podemos escrever programas que leiam o texto de volta para a memória e trabalhar com esses dados novamente.

Escrevendo dados em um arquivo vazio

Para escrever um texto em um arquivo, chame `open()` com um segundo argumento que diga a Python que você quer escrever dados no arquivo. Para ver como isso funciona, vamos escrever uma mensagem simples e armazená-la em um arquivo em vez de exibi-la na tela:

write_message.py

```
  filename = 'programming.txt'

u with open(filename, 'w') as file_object:
v     file_object.write("I love programming.")
```

A chamada a `open()` nesse exemplo tem dois argumentos u. O primeiro argumento ainda é o nome do arquivo que queremos abrir. O segundo argumento, `'w'`, diz a Python que queremos abrir o arquivo em *modo de escrita*. Podemos abrir um arquivo em *modo de leitura* (`'r'`), em *modo de escrita* (`'w'`), em *modo de concatenação* (`'a'`) ou em um modo que permita ler e escrever no arquivo (`'r+'`). Se o argumento de modo for omitido, por padrão Python abrirá o arquivo em modo somente de leitura.

A função `open()` cria automaticamente o arquivo no qual você vai escrever caso ele ainda não exista. No entanto, tome cuidado ao abrir um arquivo em modo de escrita (`'w'`) porque se o arquivo já existir, Python o apagará antes de devolver o objeto arquivo.

Em v usamos o método `write()` no objeto arquivo para escrever uma string nesse arquivo. Esse programa não tem saída no terminal, mas se

abrir o arquivo *programming.txt*, você verá uma linha:

programming.txt

```
I love programming.
```

Esse arquivo se comporta como qualquer outro arquivo de seu computador. Você pode abri-lo, escrever um novo texto nele, copiar ou colar dados e assim por diante.

NOTA Python escreve apenas strings em um arquivo-texto. Se quiser armazenar dados numéricos em um arquivo-texto, será necessário converter os dados em um formato de string antes usando a função `str()`.

Escrevendo várias linhas

A função `write()` não acrescenta nenhuma quebra de linha ao texto que você escrever. Portanto, se escrever mais de uma linha sem incluir caracteres de quebra de linha, seu arquivo poderá não ter a aparência desejada:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

Ao abrir o arquivo *programming.txt*, você verá duas linhas emendadas:

```
I love programming.I love creating new games.
```

A inclusão de quebras de linha em suas instruções `write()` faz cada string aparecer em sua própria linha:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

A saída agora aparece em linhas separadas:

```
I love programming.
```

```
I love creating new games.
```

Podemos também usar espaços, caracteres de tabulação e linhas em branco para formatar a saída, exatamente como viemos fazendo com as saídas no terminal.

Concatenando dados em um arquivo

Se quiser acrescentar conteúdos em um arquivo em vez de sobrescrever o conteúdo existente, você pode abrir o arquivo em *modo de concatenação*. Ao abrir um arquivo em modo de concatenação, Python não apagará o arquivo antes de devolver o objeto arquivo. Qualquer linha que você escrever no arquivo será adicionada no final. Se o arquivo ainda não existe, Python criará um arquivo vazio para você.

Vamos modificar *write_message.py* acrescentando alguns novos motivos pelos quais amamos programar no arquivo existente *programming.txt*:

write_message.py

```
filename = 'programming.txt'

u with open(filename, 'a') as file_object:
v     file_object.write("I also love finding meaning in large
datasets.\n")
     file_object.write("I love creating apps that can run in a
browser.\n")
```

Em u usamos o argumento `'a'` para abrir o arquivo para concatenação, em vez de sobrescrever o arquivo existente. Em v escrevemos duas linhas novas, que são acrescentadas em *programming.txt*:

programming.txt

```
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

Ao final, temos o conteúdo original do arquivo, seguido do novo conteúdo que acabamos de acrescentar.

FAÇA VOCÊ MESMO

10.3 – Convidado: Escreva um programa que pergunte o nome ao usuário. Quando ele responder, escreva o nome em um arquivo chamado *guest.txt*.

10.4 – Lista de convidados: Escreva um laço `while` que pergunte o nome aos usuários. Quando fornecerem seus nomes, apresente uma saudação na tela e acrescente uma linha que registre a visita do usuário em um arquivo chamado *guest_book.txt*. Certifique-se de que cada entrada esteja em uma nova linha do arquivo.

10.5 – Enquete sobre programação: Escreva um laço `while` que pergunte às pessoas por que elas gostam de programação. Sempre que alguém fornecer um motivo, acrescente-o em um arquivo que armazene todas as respostas.

Exceções

Python usa objetos especiais chamados *exceções* para administrar erros que surgirem durante a execução de um programa. Sempre que ocorrer um erro que faça Python não ter certeza do que deve fazer em seguida, um objeto exceção será criado. Se você escrever um código que trate a exceção, o programa continuará executando. Se a exceção não for tratada, o programa será interrompido e um *traceback,* que inclui uma informação sobre a exceção levantada, será exibido.

As exceções são tratadas com blocos `try-except`. Um bloco `try-except` pede que Python faça algo, mas também lhe diz o que deve ser feito se uma exceção for levantada. Ao usar blocos `try-except`, seus programas continuarão a executar, mesmo que algo comece a dar errado. Em vez de tracebacks, que podem ser confusos para os usuários lerem, os usuários verão mensagens de erro simpáticas escritas por você.

Tratando a exceção ZeroDivisionError

Vamos observar um erro simples, que faz Python levantar uma exceção. Provavelmente, você sabe que é impossível dividir um número por zero, mas vamos pedir que Python faça isso, de qualquer modo:

division.py

```
print(5/0)
```

É claro que Python não pode fazer essa operação, portanto veremos um traceback:

```
  Traceback (most recent call last):
    File "division.py", line 1, in <module>
      print(5/0)
u ZeroDivisionError: division by zero
```

O erro informado em u no traceback, `ZeroDivisionError`, é um objeto exceção. Python cria esse tipo de objeto em resposta a uma situação em que ele não é capaz de fazer o que lhe pedimos. Quando isso acontece, Python interrompe o programa e informa o tipo de exceção levantado. Podemos usar essa informação para modificar nosso programa. Diremos a Python o que ele deve fazer quando esse tipo de exceção ocorrer; desse modo, se ela ocorrer novamente, estaremos preparados.

Usando blocos try-except

Quando achar que um erro pode ocorrer, você poderá usar um bloco `try-except` para tratar a exceção possível de ser levantada. Dizemos a Python para tentar executar um código e lhe dizemos o que ele deve fazer caso o código resulte em um tipo particular de exceção.

Eis a aparência de um bloco `try-except` para tratar a exceção `ZeroDivisionError`:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Colocamos `print(5/0)` – a linha que causou o erro – em um bloco `try`. Se o código em um bloco `try` funcionar, Python ignorará o bloco `except`. Se o código no bloco `try` causar um erro, o interpretador procurará um bloco `except` cujo erro coincida com aquele levantado e executará o código desse bloco.

Nesse exemplo, o código no bloco `try` gera um `ZeroDivisionError`, portanto Python procura um bloco `except` que lhe diga como deve

responder. O interpretador então executa o código desse bloco e o usuário vê uma mensagem de erro simpática no lugar de um traceback:

```
You can't divide by zero!
```

Se houver mais código depois do bloco `try-except`, o programa continuará executando, pois dissemos a Python como o erro deve ser tratado. Vamos observar um exemplo em que a captura de um erro permite que um programa continue executando.

Usando exceções para evitar falhas

Tratar erros de forma correta é importante, em especial quando o programa tiver outras atividades para fazer depois que o erro ocorrer. Isso acontece com frequência em programas que pedem dados de entrada aos usuários. Se o programa responder a entradas inválidas de modo apropriado, ele poderá pedir mais entradas válidas em vez de causar uma falha.

Vamos criar uma calculadora simples que faça apenas divisões:

division.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
u     first_number = input("\nFirst number: ")
      if first_number == 'q':
          break
v     second_number = input("Second number: ")
      if second_number == 'q':
          break
w     answer = int(first_number) / int(second_number)
      print(answer)
```

Esse programa pede que o usuário forneça um primeiro número (`first_number`) u e, se o usuário não digitar *q* para sair, pede um segundo número (`second_number`) v. Então dividimos esses dois números para obter uma resposta (`answer`) w. O programa não faz nada para tratar erros,

portanto pedir que uma divisão por zero seja feita causará uma falha no programa:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

O fato de o programa falhar é ruim, mas também não é uma boa ideia deixar que os usuários vejam os tracebacks. Usuários que não sejam técnicos ficarão confusos com eles e, em um ambiente malicioso, invasores aprenderão mais do que você quer que eles saibam a partir de um traceback. Por exemplo, eles saberão o nome de seu arquivo de programa e verão uma parte de seu código que não está funcionando de forma apropriada. Às vezes, um invasor habilidoso pode usar essas informações para determinar os tipos de ataque que podem usar contra o seu código.

Bloco else

Podemos deixar esse programa mais resistente a erros colocando a linha capaz de produzir erros em um bloco `try-except`. O erro ocorre na linha que calcula a divisão, portanto é aí que colocaremos o bloco `try-except`. Esse exemplo também inclui um bloco `else`. Qualquer código que dependa do bloco `try` executar com sucesso deve ser colocado no bloco `else`:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
u   try:
```

```
        answer = int(first_number) / int(second_number)
v     except ZeroDivisionError:
        print("You can't divide by 0!")
w     else:
        print(answer)
```

Pedimos a Python para tentar concluir a operação de divisão em um bloco `try` U, que inclui apenas o código que pode causar um erro. Qualquer código que dependa do sucesso do bloco `try` é adicionado no bloco `else`. Nesse caso, se a operação de divisão for bem-sucedida, usamos o bloco `else` para exibir o resultado W.

O bloco `except` diz como Python deve responder quando um `ZeroDivisionError` ocorrer V. Se a instrução `try` não for bem-sucedida por causa de um erro de divisão por zero, mostraremos uma mensagem simpática informando o usuário de que modo esse tipo de erro pode ser evitado. O programa continua executando e o usuário jamais verá um traceback:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!

First number: 5
Second number: 2
2.5

First number: q
```

O bloco `try-except-else` funciona assim: Python tenta executar o código que está na instrução `try`. O único código que deve estar em uma instrução `try` é aquele que pode fazer uma exceção ser levantada. Às vezes, você terá um código adicional que deverá ser executado somente se o bloco `try` tiver sucesso; esse código deve estar no bloco `else`. O bloco `except` diz a Python o que ele deve fazer, caso uma determinada exceção ocorra quando ele tentar executar o código que está na instrução `try`.

Ao prever possíveis fontes de erros, podemos escrever programas robustos, que continuarão a executar mesmo quando encontrarem dados inválidos ou se depararem com recursos ausentes. Seu código será resistente a erros inocentes do usuário e a ataques maliciosos.

Tratando a exceção FileNotFoundError

Um problema comum ao trabalhar com arquivos é o tratamento de arquivos ausentes. O arquivo que você está procurando pode estar em outro lugar, o nome do arquivo pode estar escrito de forma incorreta ou o arquivo talvez simplesmente não exista. Podemos tratar todas essas situações de um modo simples com um bloco `try-except`.

Vamos tentar ler um arquivo que não existe. O programa a seguir tenta ler o conteúdo de *Alice in Wonderland* (Alice no país das maravilhas), mas não salvei o arquivo *alice.txt* no mesmo diretório em que está *alice.py*:

alice.py

```
filename = 'alice.txt'

with open(filename) as f_obj:
    contents = f_obj.read()
```

Python não é capaz de ler um arquivo ausente, portanto uma exceção é levantada:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename) as f_obj:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

A última linha do traceback informa um `FileNotFoundError`: essa é a exceção criada por Python quando não encontra o arquivo que está tentando abrir. Nesse exemplo, a função `open()` gera o erro, portanto, para tratá-lo, o bloco `try` tem início imediatamente antes da linha que contém essa função:

```
filename = 'alice.txt'

try:
```

```
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
```

Nesse exemplo, o código no bloco `try` gera um `FileNotFoundError`, portanto Python procura um bloco `except` que trate esse erro. O interpretador então executa o código que está nesse bloco e o resultado é uma mensagem de erro simpática no lugar de um traceback:

```
Sorry, the file alice.txt does not exist.
```

O programa não tem mais nada a fazer caso o arquivo não exista, portanto o código de tratamento de erro não acrescenta muito a esse programa. Vamos expandir esse exemplo e ver como o tratamento de exceções pode ajudar quando trabalhamos com mais de um arquivo.

Analisando textos

Podemos analisar arquivos-texto que contenham blocos inteiros. Muitas obras clássicas de literatura estão disponíveis como arquivos-texto simples, pois estão em domínio público. Os textos usados nesta seção foram extraídos do Projeto Gutenberg (*http://gutenberg.org/*). O Projeto Gutenberg mantém uma coleção de obras literárias disponíveis em domínio público, e é um ótimo recurso se você estiver interessado em trabalhar com textos literários em seus projetos de programação.

Vamos obter o texto de *Alice in Wonderland* e tentar contar o número de palavras do texto. Usaremos o método de string `split()`, que cria uma lista de palavras a partir de uma string. Eis o que `split()` faz com uma string que contém apenas o título `"Alice in Wonderland"`:

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

O método `split()` separa uma string em partes sempre que encontra um espaço, e armazena todas as partes da string em uma lista. O resultado é uma lista de palavras da string, embora algumas pontuações possam

também aparecer com determinadas palavras. Para contar o número de palavras em *Alice in Wonderland*, usaremos `split()` no texto todo. Em seguida, contaremos os itens da lista para ter uma ideia geral da quantidade de palavras no texto:

```
filename = 'alice.txt'

try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    # Conta o número aproximado de palavras no arquivo
u     words = contents.split()
v     num_words = len(words)
w     print("The file " + filename + " has about " + str(num_words) + "
words.")
```

Movi o arquivo *alice.txt* para o diretório correto, portanto o bloco `try` funcionará dessa vez. Em u usamos a string `contents`, que agora contém todo o texto de *Alice in Wonderland* como uma string longa, e aplicamos o método `split()` para obter uma lista de todas as palavras do livro. Quando usamos `len()` nessa lista para verificar o seu tamanho, obtemos uma boa aproximação do número de palavras na string original v. Em w exibimos uma frase que informa quantas palavras encontramos no arquivo. Esse código é colocado no bloco `else` porque funcionará somente se o código no bloco `try` for executado com sucesso. A saída nos informa quantas palavras estão em *alice.txt*:

```
The file alice.txt has about 29461 words.
```

A contagem é um pouco alta devido a informações extras fornecidas pela editora no arquivo-texto usado aqui, mas é uma boa aproximação do tamanho de *Alice in Wonderland*.

Trabalhando com vários arquivos

Vamos acrescentar outros livros para analisar. Porém, antes disso, vamos passar a parte principal desse programa para uma função chamada `count_words()`. Com isso, será mais fácil fazer a análise para diversos livros:

word_count.py

```
  def count_words(filename):
u     """Conta o número aproximado de palavras em um arquivo."""
      try:
          with open(filename) as f_obj:
              contents = f_obj.read()
      except FileNotFoundError:
          msg = "Sorry, the file " + filename + " does not exist."
          print(msg)
      else:
          # Conta o número aproximado de palavras no arquivo
          words = contents.split()
          num_words = len(words)
          print("The file " + filename + " has about " + str(num_words) +
  " words.")

  filename = 'alice.txt'
  count_words(filename)
```

A maior parte desse código não foi alterada. Simplesmente, indentamos o código e o movemos para o corpo de `count_words()`. Manter os comentários atualizados é um bom hábito quando modificamos um programa; assim, transformamos o comentário em uma docstring e o alteramos um pouco **u**.

Agora podemos escrever um laço simples para contar as palavras de qualquer texto que quisermos analisar. Fazemos isso armazenando os nomes dos arquivos que desejamos analisar em uma lista e, em seguida, chamando `count_words()` para cada arquivo da lista. Vamos experimentar contar as palavras das obras *Alice in Wonderland*, *Siddhartha*, *Moby Dick* e *Little Women*, todas disponíveis em domínio público. Deixei *siddhartha.txt* fora do diretório que contém *word_count.py* de propósito para que possamos ver como nosso programa trata um arquivo ausente de modo apropriado:

```
  def count_words(filename):
```

```
    --trecho omitido--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
'little_women.txt']
for filename in filenames:
    count_words(filename)
```

O arquivo *siddhartha.txt* ausente não tem efeito algum no restante da execução do programa:

```
The file alice.txt has about 29461 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215136 words.
The file little_women.txt has about 189079 words.
```

O uso do bloco `try-except` nesse exemplo oferece duas vantagens significativas. Evitamos que nossos usuários vejam um traceback e deixamos o programa continuar a análise dos textos que puder encontrar. Se não capturássemos e erro `FileNotFoundError` gerado por *siddhartha.txt*, o usuário veria um traceback completo e o programa pararia de executar após tentar analisar *Siddhartha*. *Moby Dick* e *Little Women* não seriam analisados.

Falhando silenciosamente

No exemplo anterior, informamos nossos usuários que um dos arquivos estava indisponível. Porém, não precisamos informar todas as exceções capturadas. Às vezes, queremos que o programa falhe silenciosamente quando uma exceção ocorrer e continue como se nada tivesse acontecido. Para fazer um programa falhar em silêncio, escreva um bloco `try` como seria feito normalmente, mas diga de forma explícita a Python para não fazer nada no bloco `except`. Python tem uma instrução `pass` que lhe diz para não fazer nada em um bloco:

```
def count_words(filename):
    """Conta o número aproximado de palavras em um arquivo."""
    try:
        --trecho omitido--
    except FileNotFoundError:
u       pass
```

```
    else:
        --trecho omitido--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
'little_women.txt']
for filename in filenames:
    count_words(filename)
```

A única diferença entre essa listagem e a listagem anterior está na instrução `pass` em **u**. Agora, quando um `FileNotFoundError` é levantado, o código no bloco `except` é executado, mas nada acontece. Nenhum traceback é gerado e não há nenhuma saída em resposta ao erro levantado. Os usuários veem os contadores de palavras para cada arquivo existente, mas não há indicação sobre um arquivo não encontrado:

```
The file alice.txt has about 29461 words.
The file moby_dick.txt has about 215136 words.
The file little_women.txt has about 189079 words.
```

A instrução `pass` também atua como um marcador. É um lembrete de que você optou por não fazer nada em um ponto específico da execução de seu programa, mas talvez queira fazer algo nesse local, no futuro. Por exemplo, nesse programa, podemos decidir escrever os nomes de qualquer arquivo ausente em um arquivo chamado *missing_files.txt*. Nossos usuários não verão esse arquivo, mas poderemos lê-lo e tratar qualquer texto ausente.

Decidindo quais erros devem ser informados

Como sabemos quando devemos informar um erro aos usuários e quando devemos falhar silenciosamente? Se os usuários souberem quais textos devem ser analisados, poderão apreciar uma mensagem que lhes informe por que alguns textos não foram analisados. Se os usuários esperam ver alguns resultados, mas não sabem quais livros deveriam ser analisados, talvez não precisem saber que alguns textos estavam indisponíveis. Dar informações que os usuários não estejam esperando pode reduzir a usabilidade de seu programa. As estruturas de tratamento de erros de Python permitem ter um controle minucioso sobre o que você deve compartilhar com os usuários quando algo sair errado; cabe a você decidir

a quantidade de informações a serem compartilhadas.

Um código bem escrito, testado de modo apropriado, não será muito suscetível a erros internos, como erros de sintaxe ou de lógica. Contudo, sempre que seu programa depender de algo externo, por exemplo, de uma entrada de usuário, da existência de um arquivo ou da disponibilidade de uma conexão de rede, existe a possibilidade de uma exceção ser levantada. Um pouco de experiência ajudará você a saber em que pontos deverá incluir blocos de tratamento de exceções em seu programa e a quantidade de informações que deverá ser fornecida aos usuários sobre os erros que ocorrerem.

FAÇA VOCÊ MESMO

10.6 – Adição: Um problema comum quando pedir entradas numéricas ocorre quando as pessoas fornecem texto no lugar de números. Ao tentar converter a entrada para um `int`, você obterá um `TypeError`. Escreva um programa que peça dois números ao usuário. Some-os e mostre o resultado. Capture o `TypeError` caso algum dos valores de entrada não seja um número e apresente uma mensagem de erro simpática. Teste seu programa fornecendo dois números e, em seguida, digite um texto no lugar de um número.

10.7 – Calculadora para adição: Coloque o código do Exercício 10.6 em um laço `while` para que o usuário possa continuar fornecendo números, mesmo se cometerem um erro e digitarem um texto no lugar de um número.

10.8 – Gatos e cachorros: Crie dois arquivos, *cats.txt* e *dogs.txt*. Armazene pelo menos três nomes de gatos no primeiro arquivo e três nomes de cachorro no segundo arquivo. Escreva um programa que tente ler esses arquivos e mostre o conteúdo do arquivo na tela. Coloque seu código em um bloco `try-except` para capturar o erro `FileNotFound` e apresente uma mensagem simpática caso o arquivo não esteja presente. Mova um dos arquivos para um local diferente de seu sistema e garanta que o código no bloco `except` seja executado de forma apropriada.

10.9 – Gatos e cachorros silenciosos: Modifique o seu bloco `except` do Exercício 10.8 para falhar silenciosamente caso um dos arquivos esteja ausente.

10.10 – Palavras comuns: Acesse o Projeto Gutenberg (*http://gutenberg.org/*) e encontre alguns textos que você gostaria de analisar. Faça download dos arquivos-texto dessas obras ou copie o texto puro de seu navegador para um arquivo-texto em seu computador.

Você pode usar o método `count()` para descobrir quantas vezes uma palavra ou

expressão aparece em uma string. Por exemplo, o código a seguir conta quantas vezes a palavra `'row'` aparece em uma string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Observe que converter a string para letras minúsculas usando `lower()` faz com que todas as formas da palavra que você está procurando sejam capturadas, independentemente do modo como elas estiverem grafadas.

Escreva um programa que leia os arquivos que você encontrou no Projeto Gutenberg e determine quantas vezes a palavra `'the'` aparece em cada texto.

Armazenando dados

Muitos de seus programas pedirão aos usuários que forneçam determinados tipos de informação. Você pode permitir que os usuários armazenem suas preferências em um jogo ou forneçam dados para uma visualização. Qualquer que seja o foco de seu programa, você armazenará as informações fornecidas pelos usuários em estruturas de dados como listas e dicionários. Quando os usuários fecham um programa, quase sempre você vai querer salvar as informações que eles forneceram. Uma maneira simples de fazer isso envolve armazenar seus dados usando o módulo `json`.

O módulo `json` permite descarregar estruturas de dados Python simples em um arquivo e carregar os dados desse arquivo na próxima vez que o programa executar. Também podemos usar `json` para compartilhar dados entre diferentes programas Python. Melhor ainda, o formato de dados JSON não é específico de Python, portanto podemos compartilhar dados armazenados em formato JSON com pessoas que trabalhem com várias outras linguagens de programação. É um formato útil e portável, além de ser fácil de aprender.

NOTA O formato JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript) foi originalmente desenvolvido para JavaScript. Apesar disso, tornou-se um formato comum, usado por muitas linguagens, incluindo Python.

Usando json.dump() e json.load()

Vamos escrever um pequeno programa que armazene um conjunto de números e outro que leia esses números de volta para a memória. O primeiro programa usará `json.dump()` para armazenar o conjunto de números, e o segundo programa usará `json.load()`.

A função `json.dump()` aceita dois argumentos: um dado para armazenar e um objeto arquivo que pode ser usado para armazenar o dado. Eis o modo como podemos usar essa função para armazenar uma lista de números:

number_writer.py

```
  import json

  numbers = [2, 3, 5, 7, 11, 13]

u filename = 'numbers.json'
v with open(filename, 'w') as f_obj:
w     json.dump(numbers, f_obj)
```

Inicialmente importamos o módulo `json` e criamos uma lista de números com a qual trabalharemos. Em u escolhemos o nome de um arquivo em que armazenaremos a lista de números. É comum usar a extensão de arquivo *.json* para indicar que os dados do arquivo estão armazenados em formato JSON. Em seguida, abrimos o arquivo em modo de escrita, o que permite a `json` escrever os dados no arquivo v. Em w usamos a função `json.dump()` para armazenar a lista `numbers` no arquivo *numbers.json*.

Esse programa não tem uma saída, mas vamos abrir o arquivo *numbers.json* e observá-lo. Os dados estão armazenados em um formato que se parece com Python:

```
[2, 3, 5, 7, 11, 13]
```

Agora escreveremos um programa que use `json.load()` para ler a lista de volta para a memória:

number_reader.py

```
  import json
```

```
u filename = 'numbers.json'
v with open(filename) as f_obj:
w     numbers = json.load(f_obj)

  print(numbers)
```

Em u garantimos que o mesmo arquivo em que escrevemos os dados será lido. Dessa vez, quando abrirmos o arquivo, fazemos isso em modo de leitura, pois Python precisará apenas ler dados do arquivo v. Em w usamos a função `json.load()` para carregar as informações armazenadas em *numbers.json* e as guardamos na variável `numbers`. Por fim, exibimos a lista de números recuperada; podemos ver que é a mesma lista criada em *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

Essa é uma maneira simples de compartilhar dados entre dois programas.

Salvando e lendo dados gerados pelo usuário

Salvar dados com `json` é conveniente quando trabalhamos com dados gerados pelo usuário porque, se você não salvar as informações de seus usuários de algum modo, elas serão perdidas quando o programa parar de executar. Vamos observar um exemplo em que pedimos aos usuários que forneçam seus nomes na primeira vez em que o programa executar e, então, o programa deverá lembrar esses nomes quando for executado novamente.

Vamos começar armazenando o nome do usuário:

remember_me.py

```
  import json

u username = input("What is your name? ")

  filename = 'username.json'
  with open(filename, 'w') as f_obj:
v     json.dump(username, f_obj)
w     print("We'll remember you when you come back, " + username + "!")
```

Em u pedimos o nome do usuário para que seja armazenado. Em seguida, usamos `json.dump()`, passando-lhe um nome de usuário e um objeto arquivo em que esse nome será armazenado v. Então exibimos uma mensagem informando o usuário que armazenamos suas informações w:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Vamos agora escrever um novo programa que faça uma saudação a um usuário cujo nome já esteja armazenado:

greet_user.py

```
  import json

  filename = 'username.json'

  with open(filename) as f_obj:
u     username = json.load(f_obj)
v     print("Welcome back, " + username + "!")
```

Em u usamos `json.load()` para ler as informações armazenadas em *username.json* na variável `username`. Agora que recuperamos o nome do usuário, podemos lhe desejar as boas-vindas de voltav:

```
Welcome back, Eric!
```

Precisamos combinar esses dois programas em um só arquivo. Quando alguém executar *remember_me.py*, queremos recuperar seu nome de usuário da memória, se for possível; assim, começaremos com um bloco `try`, que tentará recuperar o nome do usuário. Se o arquivo *username.json* não existir, faremos o bloco `except` pedir um nome de usuário e armazená-lo em *username.json* para ser usado da próxima vez:

remember_me.py

```
import json

# Carrega o nome do usuário se foi armazenado anteriormente
#  Caso contrário, pede que o usuário forneça o nome e armazena essa
informação
```

```
  filename = 'username.json'
  try:
u     with open(filename) as f_obj:
v         username = json.load(f_obj)
w except FileNotFoundError:
x     username = input("What is your name? ")
y     with open(filename, 'w') as f_obj:
          json.dump(username, f_obj)
          print("We'll remember you when you come back, " + username +
  "!")
  else:
      print("Welcome back, " + username + "!")
```

Não há nenhum código novo aqui; os blocos de código dos dois últimos exemplos simplesmente foram combinados em um só arquivo. Em u tentamos abrir o arquivo *username.json*. Se esse arquivo existir, lemos o nome do usuário de volta para a memória v e exibimos uma mensagem desejando boas-vindas de volta ao usuário no bloco `else`. Se essa é a primeira vez que o usuário executa o programa, *username.json* não existirá e um `FileNotFoundError` ocorrerá w. Python prosseguirá para o bloco `except`, em que pedimos ao usuário que forneça o seu nome x. Então usamos `json.dump()` para armazenar o nome do usuário e exibimos uma saudação y.

Qualquer que seja o bloco executado, o resultado será um nome de usuário e uma saudação apropriada. Se essa for a primeira vez que o programa é executado, a saída será assim:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Caso contrário, será:

```
Welcome back, Eric!
```

Essa é a saída que você verá se o programa já foi executado pelo menos uma vez.

Refatoração

Com frequência você chegará a um ponto em que seu código funcionará,

mas reconhecerá que ele poderia ser melhorado se fosse dividido em uma série de funções com tarefas específicas. Esse processo se chama *refatoração*. A refatoração deixa seu código mais limpo, mais fácil de compreender e de estender.

Podemos refatorar *remember_me.py* passando a maior parte de sua lógica para uma ou mais funções. O foco de *remember_me.py* está na saudação ao usuário, portanto vamos transferir todo o código existente para uma função chamada `greet_user()`:

remember_me.py

```
  import json

  def greet_user():
u     """Saúda o usuário pelo nome."""
      filename = 'username.json'
      try:
          with open(filename) as f_obj:
              username = json.load(f_obj)
      except FileNotFoundError:
          username = input("What is your name? ")
          with open(filename, 'w') as f_obj:
              json.dump(username, f_obj)
              print("We'll remember you when you come back, " + username
  + "!")
      else:
          print("Welcome back, " + username + "!")

  greet_user()
```

Como estamos usando uma função agora, atualizamos os comentários com uma docstring que reflete como o programa funciona no momento **u**. Esse arquivo é um pouco mais limpo, porém a função `greet_user()` faz mais do que simplesmente saudar o usuário – ela também recupera um nome de usuário armazenado, caso haja um, e pede que o usuário forneça um novo nome, caso não exista.

Vamos refatorar `greet_user()` para que não faça tantas tarefas diferentes. Começaremos transferindo o código para recuperar um nome de usuário já

armazenado para uma função diferente:

```
  import json

  def get_stored_username():
u     """Obtém o nome do usuário já armazenado se estiver disponível."""
      filename = 'username.json'
      try:
          with open(filename) as f_obj:
              username = json.load(f_obj)
      except FileNotFoundError:
v         return None
      else:
          return username

  def greet_user():
      """Saúda o usuário pelo nome."""
      username = get_stored_username()
w     if username:
          print("Welcome back, " + username + "!")
      else:
          username = input("What is your name? ")
          filename = 'username.json'
          with open(filename, 'w') as f_obj:
              json.dump(username, f_obj)
              print("We'll remember you when you come back, " + username
  + "!")

  greet_user()
```

A nova função `get_stored_username()` tem um propósito claro, conforme informado pela docstring em u. Essa função recupera um nome de usuário já armazenado e devolve esse nome se encontrar um. Se o arquivo *username.json* não existir, a função devolverá `None` v. Essa é uma boa prática: uma função deve devolver o valor esperado ou `None`. Isso nos permite fazer um teste simples com o valor de retorno da função. Em w exibimos uma mensagem de boas-vindas de volta ao usuário se a tentativa de recuperar um nome foi bem-sucedida; caso contrário, pedimos que um novo nome de usuário seja fornecido.

Devemos fatorar mais um bloco de código, removendo-o de `greet_user()`. Se o nome do usuário não existir, devemos transferir o código que pede um novo nome de usuário para uma função dedicada a esse propósito:

```
import json

def get_stored_username():
    """Obtém o nome do usuário já armazenado se estiver disponível."""
    --trecho omitido--

def get_new_username():
    """Pede um novo nome de usuário."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f_obj:
        json.dump(username, f_obj)
    return username

def greet_user():
    """Saúda o usuário pelo nome."""
    username = get_stored_username()
    if username:
        print("Welcome back, " + username + "!")
    else:
        username = get_new_username()
        print("We'll remember you when you come back, " + username +
"!")

greet_user()
```

Cada função nessa última versão de *remember_me.py* tem um propósito único e claro. Chamamos `greet_user()` e essa função exibe uma mensagem apropriada: ela dá as boas-vindas de volta a um usuário existente ou saúda um novo usuário. Isso é feito por meio da chamada a `get_stored_username()`, que é responsável somente por recuperar um nome de usuário já armazenado, caso exista. Por fim, `greet_user()` chama `get_new_username()` se for necessário; essa função é responsável somente por obter um novo nome de usuário e armazená-lo. Essa separação do trabalho em compartimentos é uma parte essencial na escrita de um código claro,

que seja fácil de manter e de estender.

FAÇA VOCÊ MESMO

10.11 – Número favorito: Escreva um programa que pergunte qual é o número favorito de um usuário. Use `json.dump()` para armazenar esse número em um arquivo. Escreva um programa separado que leia esse valor e apresente a mensagem "Eu sei qual é o seu número favorito! É _____.".

10.12 – Lembrando o número favorito: Combine os dois programas do Exercício 10.11 em um único arquivo. Se o número já estiver armazenado, informe o número favorito ao usuário. Caso contrário, pergunte ao usuário qual é o seu número favorito e armazene-o em um arquivo. Execute o programa duas vezes para garantir que ele funciona.

10.13 – Verificando se é o usuário correto: A última listagem de *remember_me.py* supõe que o usuário já forneceu seu nome ou que o programa está executando pela primeira vez. Devemos modificá-lo para o caso de o usuário atual não ser a pessoa que usou o programa pela última vez.

Antes de exibir uma mensagem de boas-vindas de volta em `greet_user()`, pergunte ao usuário se seu nome está correto. Se não estiver, chame `get_new_username()` para obter o nome correto.

Resumo

Neste capítulo aprendemos a trabalhar com arquivos. Vimos como ler um arquivo todo de uma só vez e como ler o conteúdo de um arquivo uma linha de cada vez. Aprendemos a escrever dados em um arquivo e a concatenar texto no final dele. Lemos sobre as exceções e o modo de tratar aquelas que provavelmente você verá em seus programas. Por fim, aprendemos a armazenar estruturas de dados Python para que possamos salvar as informações fornecidas pelos usuários, evitando que eles precisem fazer tudo de novo sempre que executarem um programa.

No Capítulo 11, conheceremos maneiras eficientes de testar o seu código. Isso ajudará você a ter mais confiança de que o código desenvolvido está correto e ajudará a identificar bugs introduzidos à medida que você continuar a estender os programas que escreveu.

11
TESTANDO O SEU CÓDIGO

Quando escrevemos uma função ou uma classe, podemos também escrever testes para esse código. Os testes provam que seu código funciona como deveria em resposta a todos os tipos de entrada para os quais ele foi projetado para receber. Ao escrever testes, você poderá estar confiante de que seu código funcionará corretamente quando mais pessoas começarem a usar seus programas. Você também poderá testar novos códigos à medida que adicioná-los para garantir que suas alterações não afetem o comportamento já existente em seu programa. Todo programador comete erros, portanto todo programador deve testar seus códigos com frequência, identificando os problemas antes que os usuários os encontrem.

Neste capítulo aprenderemos a testar o código usando ferramentas do módulo `unittest` de Python. Veremos como criar um caso de teste e verificar se um conjunto de entradas resulta na saída desejada. Conheceremos a aparência de um teste que passa e de um teste que não passa, e veremos como um teste que falha pode nos ajudar a melhorar o

código. Aprenderemos a testar funções e classes, e você começará a entender quantos testes devem ser escritos para um projeto.

Testando uma função

Para aprender a testar, precisamos de um código para testes. Eis uma função simples que aceita um primeiro nome e um sobrenome e devolve um nome completo formatado de modo elegante:

name_function.py

```
def get_formatted_name(first, last):
    """Gera um nome completo formatado de modo elegante."""
    full_name = first + ' ' + last
    return full_name.title()
```

A função `get_formatted_name()` combina o primeiro nome e o sobrenome com um espaço entre eles para compor um nome completo e então converte as primeiras letras do nome para maiúsculas e devolve o nome completo. Para verificar se `get_formatted_name()` funciona, vamos criar um programa que use essa função. O programa *names.py* permite que os usuários forneçam um primeiro nome e um sobrenome e vejam um nome completo formatado de modo elegante:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print("\tNeatly formatted name: " + formatted_name + '.')
```

Esse programa importa `get_formatted_name()` de *name_function.py*. O usuário pode fornecer uma série de primeiros nomes e de sobrenomes e ver os nomes completos formatados:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
        Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
        Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

Podemos ver que os nomes gerados nesse caso estão corretos. Porém, vamos supor que queremos modificar `get_formatted_name()` para que ele seja capaz de lidar com nomes do meio também. Quando fizermos isso, queremos ter certeza de que não causaremos erros no modo como a função trata os nomes que tenham apenas um primeiro nome e um sobrenome. Poderíamos testar nosso código executando *names.py* e fornecendo um nome como `Janis Joplin` sempre que modificarmos `get_formatted_name()`, mas isso seria tedioso. Felizmente Python oferece um modo eficiente de automatizar os testes da saída de uma função. Se os testes de `get_formatted_name()` forem automatizados, poderemos sempre ter a confiança de que a função estará correta quando fornecermos os tipos de nomes para os quais os testes forem escritos.

Testes de unidade e casos de teste

O módulo `unittest` da biblioteca-padrão de Python oferece as ferramentas para testar seu código. Um *teste de unidade* verifica se um aspecto específico do comportamento de uma função está correto. Um *caso de teste* é uma coleção de testes de unidade que, em conjunto, prova que uma função se comporta como deveria em todas as situações que você espera que ela trate. Um bom caso de teste considera todos os tipos possíveis de entradas que uma função poderia receber e inclui testes para representar

cada uma dessas situações. Um caso de teste com *cobertura completa* é composto de uma variedade de testes de unidade que inclui todas as possíveis maneiras de usar uma função. Atingir a cobertura completa em um projeto de grande porte pode ser desanimador. Em geral, é suficiente escrever testes para os comportamentos críticos de seu código e então visar a uma cobertura completa somente se o projeto começar a ter uso disseminado.

Um teste que passa

A sintaxe para criar um caso de teste exige um pouco de prática, mas depois que você o configurar, será mais fácil adicionar outros casos de teste para suas funções. Para escrever um caso de teste para uma função, importe o módulo `unittest` e a função que você quer testar. Em seguida crie uma classe que herde de `unittest.TestCase` e escreva uma série de métodos para testar diferentes aspectos do comportamento de sua função.

Eis um caso de teste com um método que verifica se a função `get_formatted_name()` está correta quando recebe um primeiro nome e um sobrenome:

test_name_function.py

```
  import unittest
  from name_function import get_formatted_name

u class NamesTestCase(unittest.TestCase):
      """Testes para 'name_function.py'."""

      def test_first_last_name(self):
          """Nomes como 'Janis Joplin' funcionam?"""
v         formatted_name = get_formatted_name('janis', 'joplin')
w         self.assertEqual(formatted_name, 'Janis Joplin')

  unittest.main()
```

Inicialmente importamos `unittest` e a função `get_formatted_name()` que queremos testar. Em u criamos uma classe chamada `NamesTestCase`, que conterá uma série de testes de unidade para `get_formatted_name()`. Você

pode dar o nome que quiser para a classe, mas é melhor nomeá-la com palavras relacionadas à função que você está prestes a testar e usar a palavra *Test* no nome da classe. Essa classe dever herdar da classe `unittest.TestCase` para que Python saiba executar os testes que você escrever.

`NamesTestCase` contém um único método que testa um aspecto de `get_formatted_name()`. Chamamos esse método de `test_first_last_name()` porque estamos verificando se os nomes que têm apenas o primeiro nome e o sobrenome são formatados corretamente. Qualquer método que comece com `test_` será executado de modo automático quando *test_name_function.py* for executado. Nesse método de teste, chamamos a função que queremos testar e armazenamos um valor de retorno que estamos interessados em testar. Nesse exemplo, chamamos `get_formatted_name()` com os argumentos `'janis'` e `'joplin'` e armazenamos o resultado em `formatted_name` V.

Em W usamos um dos recursos mais úteis de `unittest`: um método de *asserção*. Os métodos de asserção verificam se um resultado recebido é igual ao resultado que você esperava receber. Nesse caso, como sabemos que `get_formatted_name()` deve devolver um nome completo, com as letras iniciais maiúsculas e os espaços apropriados, esperamos que o valor em `formatted_name` seja `Janis Joplin`. Para conferir se isso é verdade, usamos o método `assertEqual()` de `unittest` e lhe passamos `formatted_name` e `'Janis Joplin'`. A linha

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

diz o seguinte: "Compare o valor em `formatted_name` com a string `'Janis Joplin'`. Se forem iguais conforme esperado, tudo bem. Contudo, se não forem iguais, me avise!".

A linha `unittest.main()` diz a Python para executar os testes desse arquivo. Quando executamos *test_name_function.py*, vemos a saída a seguir:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

```
OK
```

O ponto na primeira linha da saída nos informa que um único teste passou. A próxima linha diz que Python executou um teste e demorou menos de 0,001 segundo para fazê-lo. O `OK` no final informa que todos os testes de unidade do caso de teste passaram.

Essa saída mostra que a função `get_formatted_name()` sempre funcionará para nomes que tenham o primeiro nome e o sobrenome, a menos que a função seja modificada. Se modificarmos `get_formatted_name()`, poderemos executar esse teste novamente. Se o caso de teste passar, saberemos que a função continua funcionando para nomes como Janis Joplin.

Um teste que falha

Como é a aparência de um teste que falha? Vamos modificar `get_formatted_name()` para que possa tratar nomes do meio, mas faremos isso de modo que a função gere um erro para nomes que tenham apenas um primeiro nome e um sobrenome, como Janis Joplin.

A seguir, apresentamos uma nova versão de `get_formatted_name()` que exige um argumento para um nome do meio:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Gera um nome completo formatado de modo elegante."""
    full_name = first + ' ' + middle + ' ' + last
    return full_name.title()
```

Essa versão deve funcionar para pessoas com nomes do meio, mas quando a testamos, vemos que ela deixou de funcionar para pessoas que tenham apenas um primeiro nome e um sobrenome. Dessa vez, a execução do arquivo *test_name_function.py* fornece o resultado a seguir:

```
u E
  ======================================================================
v ERROR: test_first_last_name (__main__.NamesTestCase)
  ----------------------------------------------------------------------
w Traceback (most recent call last):
```

```
    File "test_name_function.py", line 8, in test_first_last_name
      formatted_name = get_formatted_name('janis', 'joplin')
  TypeError: get_formatted_name() missing 1 required positional argument:
  'last'

  ----------------------------------------------------------------------
x Ran 1 test in 0.000s

y FAILED (errors=1)
```

Há muitas informações aqui, pois há muitos dados que você precisa saber quando um teste falha. O primeiro item da saída é um único `E` u, que nos informa que um teste de unidade do caso de teste resultou em erro. A seguir, vemos que `test_first_last_name()` em `NamesTestCase` causou um erro v. Saber qual teste falhou será crucial quando o seu caso de teste tiver muitos testes de unidade. Em w vemos um traceback padrão, que informa que a chamada de função `get_formatted_name('janis', 'joplin')` não funciona mais, pois um argumento posicional obrigatório está ausente.

Também podemos ver que um único teste de unidade foi executado x. Por fim, vemos uma mensagem adicional informando que o caso de teste como um todo falhou e que houve um erro em sua execução y. Essa informação aparece no final da saída para que possa ser vista de imediato; você não vai querer fazer uma rolagem para cima em uma listagem longa de saída para descobrir quantos testes falharam.

Respondendo a um teste que falhou

O que devemos fazer quando um teste falha? Supondo que você esteja verificando as condições corretas, um teste que passa significa que a função está se comportando de forma apropriada e um teste que falha quer dizer que há um erro no novo código que você escreveu. Assim, se um teste falhar, não mude o teste. Em vez disso, corrija o código que fez o teste falhar. Analise as alterações que você acabou de fazer na função e descubra como elas afetaram o comportamento desejado.

Nesse caso, `get_formatted_name()` costumava exigir apenas dois parâmetros: um primeiro nome e um sobrenome. Agora ela exige um primeiro nome, um nome do meio e um sobrenome. A adição do

parâmetro obrigatório para o nome do meio fez o comportamento desejado de `get_formatted_name()` apresentar problemas. A melhor opção nesse caso é deixar o nome do meio opcional. Feito isso, nosso teste para nomes como `Janis Joplin` deverá passar novamente e poderemos aceitar nomes do meio também. Vamos modificar `get_formatted_name()` de modo que os nomes do meio sejam opcionais e então executar o caso de teste novamente. Se o teste passar, prosseguiremos para garantir que a função trate os nomes do meio de forma apropriada.

Para deixar os nomes do meio opcionais, passamos o parâmetro `middle` para o final da lista de parâmetros na definição da função e lhe fornecemos um valor default vazio. Além disso, acrescentamos um teste `if` que compõe o nome completo de forma apropriada, conforme um nome do meio tenha sido fornecido ou não:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Gera um nome completo formatado de modo elegante."""
    if middle:
        full_name = first + ' ' + middle + ' ' + last
    else:
        full_name = first + ' ' + last
    return full_name.title()
```

Nessa nova versão de `get_formatted_name()`, o nome do meio é opcional. Se um nome do meio for passado para a função (`if middle:`), o nome completo conterá um primeiro nome, um nome do meio e um sobrenome. Caso contrário, o nome completo será constituído apenas de um primeiro nome e de um sobrenome. Agora a função deve estar adequada para trabalhar com os dois tipos de nomes. Para descobrir se a função continua apropriada para nomes como `Janis Joplin`, vamos executar *test_name_function.py* novamente:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

O caso de teste agora passou. É a situação ideal: quer dizer que a função está correta para nomes como Janis Joplin de novo, sem que tenhamos que testar a função manualmente. Corrigir nossa função foi fácil porque o teste que falhou nos ajudou a identificar o novo código que interferiu no comportamento existente.

Adicionando novos testes

Agora que sabemos que get_formatted_name() funciona para nomes simples novamente, vamos escrever um segundo teste para pessoas que tenham um nome do meio. Fazemos isso adicionando outro método à classe NamesTestCase:

```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """Testes para 'name_function.py'."""

    def test_first_last_name(self):
        """Nomes como 'Janis Joplin' funcionam?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

    def test_first_last_middle_name(self):
        """Nomes como 'Wolfgang Amadeus Mozart' funcionam?"""
u       formatted_name = get_formatted_name(
            'wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

unittest.main()
```

Chamamos esse novo método de test_first_last_middle_name(). O nome do método deve começar com *test_* para que seja executado automaticamente quando *test_name_function.py* for executado. Nomeamos o método de modo a deixar claro qual é o comportamento de get_formatted_name() que estamos testando. Como resultado, se o teste falhar, saberemos de imediato quais tipos de nome serão afetados. Nomes longos para os métodos em nossas classes TestCase não são um problema.

Eles devem ser descritivos para que você possa compreender a saída quando seus testes falharem, e pelo fato de Python os chamar automaticamente, você não precisará escrever código para chamar esses métodos.

Para testar a função, chamamos `get_formatted_name()` com um primeiro nome, um sobrenome e um nome do meio ❶ e, em seguida, usamos `assertEqual()` para conferir se o nome completo devolvido coincide com o nome completo (primeiro nome, nome do meio e sobrenome) esperado. Se executarmos *test_name_function.py* novamente, veremos que os dois testes passam:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Ótimo! Agora sabemos que a função continua correta para nomes como `Janis Joplin`, e podemos estar confiantes de que ela funcionará para nomes como `Wolfgang Amadeus Mozart` também.

FAÇA VOCÊ MESMO

11.1 – Cidade, país: Escreva uma função que aceite dois parâmetros: o nome de uma cidade e o nome de um país. A função deve devolver uma única string no formado ***`Cidade, País`***, por exemplo, **`Santiago, Chile`**. Armazene a função em um módulo chamado *city_functions.py*.

Crie um arquivo de nome *test_cities.py* que teste a função que você acabou de escrever (lembre-se de que é necessário importar **`unittest`** e a função que você quer testar). Escreva um método chamado **`test_city_country()`** para conferir se a chamada à sua função com valores como **`'santiago'`** e **`'chile'`** resulta na string correta. Execute *test_cities.py* e garanta que **`test_city_country()`** passe no teste.

11.2 – População: Modifique sua função para que ela exija um terceiro parâmetro, **`population`**. Agora ela deve devolver uma única string no formato ***`Cidade, País – população xxx`***, por exemplo, **`Santiago, Chile – população 5000000`**. Execute *test_cities.py* novamente. Certifique-se de que **`test_city_country()`** falhe dessa vez.

Modifique a função para que o parâmetro **`population`** seja opcional. Execute

test_cities.py novamente e garanta que `test_city_country()` passe novamente.

Escreva um segundo teste chamado `test_city_country_population()` que verifique se você pode chamar sua função com os valores `'santiago'`, `'chile'` e `'population=5000000'`. Execute *test_cities.py* novamente e garanta que esse novo teste passe.

Testando uma classe

Na primeira parte deste capítulo escrevemos testes para uma única função. Agora vamos escrever testes para uma classe. Você usará classes em muitos de seus próprios programas, portanto é conveniente ser capaz de provar que suas classes funcionam corretamente. Se os testes para uma classe com a qual você estiver trabalhando passarem, você poderá estar confiante de que as melhorias que fizer nessa classe não causarão falhas por acidente no comportamento atual.

Uma variedade de métodos de asserção

Python disponibiliza vários métodos de asserção na classe `unittest.TestCase`. Como mencionamos, os métodos de asserção testam se uma condição que você acredita ser verdadeira em um ponto específico de seu código é realmente verdadeira. Se a condição for verdadeira conforme esperado, sua pressuposição sobre o comportamento dessa parte do programa será confirmada; você pode estar confiante de que não há erros. Se a condição que você supõe ser verdadeira na verdade não for, Python levantará uma exceção.

A Tabela 11.1 descreve seis métodos de asserção comumente usados. Com esses métodos, podemos verificar se os valores devolvidos são ou não são iguais aos valores esperados, se os valores são `True` ou `False` e se os valores estão (`in`) ou não estão (`not in`) em uma dada lista. Você pode usar esses métodos somente em uma classe que herde de `unittest.TestCase`, portanto vamos observar como um desses métodos pode ser usado no contexto dos testes de uma classe propriamente dita.

Tabela 11.1 – Métodos de asserção disponíveis no módulo `unittest`

Método	Uso

assertEqual(a, b)	Verifica se a == b
assertNotEqual(a, b)	Verifica se a != b
assertTrue(x)	Verifica se x é True
assertFalse(x)	Verifica se x é False
assertIn(*item*, *lista*)	Verifica se *item* está em *lista*
assertNotIn(*item*, *lista*)	Verifica se *item* não está em *lista*

Uma classe para testar

Testar uma classe é semelhante a testar uma função – boa parte de seu trabalho envolve testar o comportamento dos métodos da classe. Porém, há algumas diferenças, portanto vamos escrever uma classe para ser testada. Considere uma classe que ajude a administrar pesquisas anônimas:

survey.py

```
  class AnonymousSurvey():
      """Coleta respostas anônimas para uma pergunta de uma pesquisa."""

u     def __init__(self, question):
          """Armazena uma pergunta e se prepara para armazenar as
  respostas."""
          self.question = question
          self.responses = []

v     def show_question(self):
          """Mostra a pergunta da pesquisa."""
          print(question)

w     def store_response(self, new_response):
          """Armazena uma única resposta da pesquisa."""
          self.responses.append(new_response)

x     def show_results(self):
          """Mostra todas as respostas dadas."""
          print("Survey results:")
          for response in responses:
              print('- ' + response)
```

Essa classe começa com uma pergunta fornecida por você para uma pesquisa `u` e inclui uma lista vazia para armazenar as respostas. A classe tem métodos para exibir a pergunta da pesquisa `v`, adicionar uma nova resposta à lista de respostas `w` e exibir todas as respostas armazenadas na lista `x`. Para criar uma instância dessa classe, tudo que precisamos fazer é fornecer uma pergunta. Depois de ter criado uma instância que represente uma pesquisa em particular, você mostrará a pergunta da pesquisa com `show_question()`, armazenará uma resposta com `store_response()` e exibirá o resultado com `show_results()`.

Para mostrar que a classe `AnonymousSurvey` funciona, vamos escrever um programa que utilize essa classe:

language_survey.py

```
from survey import AnonymousSurvey

# Define uma pergunta e cria uma pesquisa
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Mostra a pergunta e armazena as respostas à pergunta
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response(response)

# Exibe os resultados da pesquisa
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

Esse programa define uma pergunta (`"What language did you first learn to speak?"`, ou seja, "Qual foi a primeira língua que você aprendeu a falar?") e cria um objeto `AnonymousSurvey` com essa pergunta. O programa chama `show_question()` para exibir a pergunta e então espera as respostas. Cada resposta é armazenada à medida que é recebida. Quando todas as

respostas tiverem sido fornecidas (o usuário digitou `q` para sair), `show_results()` exibirá o resultado da pesquisa:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

Essa classe funciona para uma pesquisa anônima simples. No entanto, vamos supor que queremos aperfeiçoar `AnonymousSurvey` e o módulo em que ele se encontra, que é `survey`. Poderíamos permitir que cada usuário forneça mais de uma resposta. Poderíamos escrever um método para listar apenas as respostas únicas e informar quantas vezes cada resposta foi dada. Também poderíamos escrever outra classe para administrar pesquisas não anônimas.

O comportamento atual da classe `AnonymousSurvey` correria o risco de ser afetado com a implementação de mudanças como essas. Por exemplo, é possível que, na tentativa de permitir que cada usuário forneça várias respostas, poderíamos acidentalmente mudar o modo como as respostas únicas são tratadas. Para garantir que não causaremos problemas no comportamento existente à medida que desenvolvemos esse módulo, podemos escrever testes para a classe.

Testando a classe AnonymousSurvey

Vamos escrever um teste que verifique um aspecto do comportamento de `AnonymousSurvey`. Escreveremos um teste para verificar se uma resposta

única à pergunta da pesquisa é armazenada de forma apropriada. Usaremos o método `assertIn()` para conferir se a resposta está na lista de respostas depois que ela for armazenada:

test_survey.py

```
  import unittest
  from survey import AnonymousSurvey

u class TestAnonmyousSurvey(unittest.TestCase):
      """Testes para a classe AnonymousSurvey"""

v     def test_store_single_response(self):
          """Testa se uma única resposta é armazenada de forma
  apropriada."""
          question = "What language did you first learn to speak?"
w         my_survey = AnonymousSurvey(question)
          my_survey.store_response('English')

x         self.assertIn('English', my_survey.responses)

  unittest.main()
```

Começamos importando o módulo `unittest` e a classe que queremos testar, isto é, `AnonymousSurvey`. Chamamos nosso caso de teste de `TestAnonymousSurvey` que, novamente, herda de `unittest.TestCase` **u**. O primeiro método de teste verificará se quando armazenamos uma resposta à pergunta da pesquisa, ela será inserida na lista de respostas da pesquisa. Um bom nome descritivo para esse método é `test_store_single_response()` **v**. Se esse teste falhar, pelo nome do método mostrado na saída do teste que falhou saberemos que houve um problema na armazenagem de uma única resposta à pesquisa.

Para testar o comportamento de uma classe, precisamos criar uma instância dessa classe. Em **w** criamos uma instância chamada `my_survey` com a pergunta `"What language did you first learn to speak?"`. Armazenamos uma única resposta, `English`, usando o método `store_response()`. Então conferimos se a resposta foi armazenada corretamente confirmando se `English` está na lista `my_survey.responses` **x**.

Quando executamos *test_survey.py*, vemos que o teste passa:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Isso é bom, mas uma pesquisa será útil somente se gerar mais de uma resposta. Vamos verificar se três respostas podem ser armazenadas corretamente. Para isso, adicionamos outro método em `TestAnonymousSurvey`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Testes para a classe AnonymousSurvey"""

    def test_store_single_response(self):
        """Testa se uma única resposta é armazenada de forma
apropriada."""
        --trecho omitido--

    def test_store_three_responses(self):
        """Testa se três respostas individuais são armazenadas de forma
apropriada."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
u       responses = ['English', 'Spanish', 'Mandarin']
        for response in responses:
            my_survey.store_response(response)

v       for response in responses:
            self.assertIn(response, my_survey.responses)

unittest.main()
```

Chamamos o novo método de `test_store_three_responses()`. Criamos um objeto para a pesquisa, exatamente como fizemos em `test_store_single_response()`. Definimos uma lista contendo três respostas

diferentes u e, então, chamamos `store_response()` para cada uma dessas respostas. Depois que as respostas foram armazenadas, escrevemos outro laço e conferimos se cada resposta está agora em `my_survey.responses` v.

Quando executamos *test_survey.py* novamente, vemos que os dois testes (para uma única resposta e para três respostas) passam:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Isso funciona perfeitamente. Porém, esses testes são um pouco repetitivos, então usaremos outro recurso de `unittest` para deixá-los mais eficientes.

Método setUp()

Em *test_survey.py* criamos uma nova instância de `AnonymousSurvey` em cada método de teste e criamos novas respostas para cada método. A classe `unittest.TestCase` tem um método `setUp()` que permite criar esses objetos uma vez e então usá-los em cada um de seus métodos de teste. Quando um método `setUp()` é incluído em uma classe `TestCase`, Python executa esse método antes de qualquer método cujo nome comece com *test_*. Qualquer objeto criado no método `setUp()` estará disponível a todos os métodos de teste que você escrever.

Vamos usar `setUp()` para criar uma instância de pesquisa e um conjunto de respostas que possa ser usado em `test_store_single_response()` e em `test_store_three_responses()`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Testes para a classe AnonymousSurvey."""

    def setUp(self):
        """
```

```
        Cria uma pesquisa e um conjunto de respostas que poderão ser
  usados em todos os métodos de teste.
        """
        question = "What language did you first learn to speak?"
u       self.my_survey = AnonymousSurvey(question)
v       self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Testa se uma única resposta é armazenada de forma
  apropriada."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Testa se três respostas individuais são armazenadas de forma
  apropriada."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

  unittest.main()
```

O método `setUp()` faz duas tarefas: cria uma instância da pesquisa **u** e cria uma lista de respostas **v**. Cada um desses dados é prefixado com `self` para que possam ser usados em qualquer lugar na classe. Isso simplifica os dois métodos de teste, pois nenhum deles precisará criar uma instância da pesquisa ou uma resposta. O método `test_store_single_response()` verifica se a primeira resposta em `self.responses` – `self.responses[0]` – pode ser armazenada corretamente, e `test_store_single_response()` verifica se todas as três respostas em `self.responses` podem ser armazenadas corretamente.

Quando executamos *test_survey.py* de novo, vemos que os dois testes continuam passando. Esses testes seriam particularmente úteis se tentássemos expandir `AnonymousSurvey` de modo a tratar várias respostas para cada pessoa. Depois de modificar o código para que aceite várias respostas, você poderia executar esses testes e garantir que não afetou a capacidade de armazenar uma única resposta ou uma série de respostas individuais.

Quando testar suas próprias classes, o método `setUp()` poderá facilitar a escrita de seus métodos de teste. Crie apenas um conjunto de instâncias e de atributos em `setUp()` e então utilize essas instâncias em todos os seus métodos de teste. Isso é muito mais fácil que criar um novo conjunto de instâncias e de atributos em cada método de teste.

NOTA Durante a execução de um caso de teste, Python exibe um caractere para cada teste de unidade à medida que ele terminar. Um teste que passar exibe um ponto, um teste que resulte em erro exibe um `E` e um teste que resultar em uma asserção com falha exibe um `F`. É por isso que você verá um número diferente de pontos e de caracteres na primeira linha da saída quando executar seus casos de teste. Se um caso de teste demorar muito para executar por conter muitos testes de unidade, você poderá observar esses resultados para ter uma noção de quantos testes estão passando.

FAÇA VOCÊ MESMO

11.3 – Funcionário: Escreva uma classe chamada `Employee`. O método `__init__()` deve aceitar um primeiro nome, um sobrenome e um salário anual, e deve armazenar cada uma dessas informações como atributos. Escreva um método de nome `give_raise()` que some cinco mil dólares ao salário anual, por default, mas que também aceite um valor diferente de aumento.

Escreva um caso de teste para `Employee`. Crie dois métodos de teste, `test_give_default_raise()` e `test_give_custom_raise()`. Use o método `setUp()` para que não seja necessário criar uma nova instância de funcionário em cada método de teste. Execute seu caso de teste e certifique-se de que os dois testes passem.

Resumo

Neste capítulo aprendemos a escrever testes para funções e classes usando ferramentas do módulo `unittest`. Vimos como escrever uma classe que herde de `unittest.TestCase` e aprendemos a escrever métodos de teste para conferir se suas funções e classes exibem comportamentos específicos. Vimos como usar o método `setUp()` para criar instâncias e atributos de modo eficiente para suas classes; assim, esses dados podem ser usados por todos os métodos de teste de uma classe.

Testar é um assunto importante, que muitos iniciantes não aprendem a

fazer. Você não precisa escrever testes para todos os projetos simples que experimentar criar como iniciante. Porém, assim que começar a trabalhar com projetos que envolvam um esforço significativo de desenvolvimento, você deve testar os comportamentos críticos de suas funções e classes. Você estará mais seguro de que novos trabalhos feitos em seu projeto não causarão falhas nas partes que funcionam, e isso lhe dará liberdade para fazer melhorias em seu código. Se você acidentalmente causar problemas em funcionalidades existentes, saberá de imediato e poderá corrigi-los com facilidade. Responder a um teste que falhou é muito mais fácil que responder a um relatório de bug de um usuário insatisfeito.

Outros programadores respeitarão mais os seus projetos se você incluir alguns testes iniciais. Eles se sentirão mais à vontade para fazer experimentos com o seu código e estarão mais dispostos a trabalhar com você nos projetos. Se quiser fazer contribuições a um projeto em que outros programadores estão trabalhando, é esperado que você mostre que seu código passe nos testes existentes e que, de modo geral, escreva testes para os novos comportamentos que introduzir no projeto.

Brinque com os testes para ter familiaridade com o processo de testar o seu código. Escreva testes para os comportamentos mais críticos de suas funções e classes, mas não vise a uma cobertura completa no início de seus projetos, a menos que você tenha um motivo específico para fazer isso.

PARTE II

PROJETOS

Parabéns! Agora você conhece Python o suficiente para começar a desenvolver projetos interativos e significativos. Criar seus próprios projetos permitirá que você adquira novas habilidades e solidificará sua compreensão dos conceitos apresentados na Parte I.

A Parte II contém três tipos de projeto, e você pode optar por desenvolver qualquer um desses projetos, ou todos eles, na ordem que quiser. Eis uma rápida descrição de cada projeto para ajudar a decidir qual deles você explorará antes.

Invasão Alienígena: criando um jogo com Python

No projeto da Invasão Alienígena (Capítulos 12, 13 e 14), usaremos o pacote Pygame para desenvolver um jogo 2D em que o objetivo é atirar em uma frota de alienígenas à medida que caem na tela, em níveis crescentes de velocidade e de dificuldade. No final do projeto, você terá adquirido habilidades que permitirão o desenvolvimento de seus próprios jogos 2D com o Pygame.

Visualização de dados

O projeto de Visualização de Dados começa no Capítulo 15, em que você aprenderá a gerar dados e a criar uma série de visualizações funcionais e bonitas desses dados usando matplotlib e Pygal. O Capítulo 16 ensina você a acessar dados de fontes online e a fornecê-los a um pacote de visualização para criar apresentações de dados meteorológicos e um mapa da população

mundial. Por fim, o Capítulo 17 mostra como escrever um programa para fazer download automático de dados e visualizá-los. Aprender a criar visualizações permite explorar o campo do data mining (mineração de dados), que é uma habilidade bastante procurada no mundo de hoje.

Aplicações web

No projeto de Aplicações Web (Capítulos 18, 19 e 20), você usará o pacote Django para criar uma aplicação web simples que permita aos usuários manter um diário sobre qualquer assunto que estejam aprendendo. Os usuários criarão uma conta com um nome de usuário e uma senha, fornecerão um assunto e então criarão entradas sobre o que estiverem aprendendo. Também veremos como fazer a implantação de suas aplicações para que qualquer pessoa no mundo possa acessá-las.

Depois de concluir esse projeto, você será capaz de iniciar suas próprias aplicações web simples e estará pronto para mergulhar de cabeça em recursos mais abrangentes para o desenvolvimento de aplicações com Django.

PROJETO 1

INVASÃO ALIENÍGENA

12

UMA ESPAÇONAVE QUE ATIRA

Vamos criar um jogo! Usaremos o Pygame – uma coleção de módulos Python divertida e eficaz que administra imagens gráficas, animações e até mesmo sons, facilitando o desenvolvimento de jogos sofisticados. Com o Pygame tratando tarefas como desenhar imagens na tela, você poderá ignorar boa parte do código tedioso e difícil e se concentrar na lógica de mais alto nível da dinâmica dos jogos.

Neste capítulo instalaremos o Pygame e então criaremos uma espaçonave que se move para a direita e para a esquerda e atira em resposta à entrada do usuário. Nos próximos dois capítulos, criaremos uma frota de alienígenas para destruir e então continuaremos a fazer ajustes finos, por exemplo, definindo limites para o número de espaçonaves que poderão ser usadas e acrescentando uma tabela de pontuação.

Neste capítulo você aprenderá também a administrar projetos grandes, que contêm vários arquivos. Faremos a refatoração de vários códigos e administraremos o conteúdo dos arquivos para manter nosso projeto organizado e o código eficiente.

Desenvolver jogos é um modo ideal de se divertir ao mesmo tempo que aprendemos uma linguagem. É extremamente satisfatório ver outros jogadores usarem um jogo escrito por você mesmo, e escrever um jogo simples o ajudará a entender como os jogos profissionais são criados. À medida que trabalhar neste capítulo, digite e execute o código para entender como cada bloco de código contribui com o gameplay[1] em geral. Faça experimentos com valores e configurações diferentes para compreender melhor de que modo você pode sofisticar mais as interações em seus próprios jogos.

NOTA A Invasão Alienígena ocupará vários arquivos diferentes, portanto crie uma nova pasta em seu sistema chamada *alien_invasion*. Não se esqueça de salvar

todos os arquivos do projeto nessa pasta para que suas instruções `import` funcionem corretamente.

Planejando o seu projeto

Ao desenvolver um projeto grande, é importante preparar um plano antes de começar a escrever o seu código. Seu plano manterá você focado e fará com que seja mais provável que o projeto seja concluído.

Vamos escrever uma descrição do gameplay como um todo. Embora a descrição a seguir não inclua todos os detalhes da Invasão Alienígena, ela oferece uma ideia clara de como podemos começar a desenvolver o jogo:

> *Na Invasão Alienígena, o jogador controla uma espaçonave que aparece na parte inferior central da tela. O jogador pode mover a espaçonave para a direita e para a esquerda usando as teclas de direção e atirar usando a barra de espaço. Quando o jogo começa, uma frota de alienígenas enche o céu e se desloca na tela para os lados e para baixo. O jogador atira nos alienígenas e os destrói. Se o jogador atingir todos os alienígenas, uma nova frota, que se moverá mais rapidamente que a frota anterior, aparecerá. Se algum alienígena atingir a espaçonave do jogador ou alcançar a parte inferior da tela, o jogador perderá uma nave. Se o jogador perder três espaçonaves, o jogo terminará.*

Na primeira fase do desenvolvimento criaremos uma espaçonave capaz de se mover para a direita e para a esquerda. A nave deverá ser capaz de atirar quando o jogador pressionar a barra de espaço. Depois de proporcionar esse comportamento, poderemos voltar nossa atenção aos alienígenas e sofisticar o gameplay.

Instalando o Pygame

Antes de começar a programar, instale o Pygame. A seguir, descreveremos como isso é feito no Linux, no OS X e no Microsoft Windows.

Se você usa Python 3 no Linux, ou utiliza o OS X, será necessário usar o pip para instalar o Pygame. O pip é um programa que cuida do download e da instalação de pacotes Python para você. As próximas seções mostrarão

como instalar pacotes usando o pip.

Se você usa Python 2.7 no Linux ou utiliza o Windows, não será necessário usar o pip para instalar o Pygame. Em vez disso, vá para a seção “Instalando o Pygame no Linux”, ou para a seção “Instalando o Pygame no Windows”.

NOTA As instruções para instalar o pip em todos os sistemas estão nas próximas seções, pois você precisará do pip para os projetos de visualização de dados e de aplicações web. Essas instruções também estão incluídas nos recursos online em *https://www.nostarch.com/pythoncrashcourse/*. Se tiver problemas com as instruções aqui, verifique se as instruções online são mais adequadas a você.

Instalando pacotes Python com o pip

As versões mais recentes de Python vêm com pip instalado, portanto verifique antes se ele já está em seu sistema. Em Python 3, o pip às vezes é chamado de *pip3*.

Verificando se o pip está instalado no Linux e no OS X

Abra uma janela do terminal e digite o seguinte comando:

```
  $ pip --version
u pip 7.0.3 from /usr/local/lib/python3.5/dist-packages (python 3.5)
  $
```

Se você tiver apenas uma versão de Python instalada em seu sistema e vir uma saída semelhante a essa, vá para a seção “Instalando o Pygame no Linux”, ou para a seção “Instalando o Pygame no OS X”. Se vir uma mensagem de erro, tente usar `pip3` no lugar de `pip`. Se nenhuma das versões estiver instalada em seu sistema, vá para a seção “Instalando o pip”.

Se houver mais de uma versão de Python em seu sistema, verifique se o pip está associado à versão de Python que você está usando – por exemplo, `python 3.5` em u. Se o pip estiver associado à versão correta de Python, vá para a seção “Instalando o Pygame no Linux”, ou para a seção “Instalando o Pygame no OS X”. Se o pip estiver associado à versão incorreta de Python, experimente usar `pip3` no lugar de `pip`. Se nenhum dos comandos funcionar para a versão de Python que você está usando, vá para a seção “Instalando o pip”.

Verificando se o pip está instalado no Windows

Abra uma janela do terminal e digite o seguinte comando:

```
  $ python -m pip --version
u pip 7.0.3 from C:\Python35\lib\site-packages (python 3.5)
  $
```

Se seu sistema tiver apenas uma versão de Python instalada e você vir uma saída semelhante a essa, vá para a seção "Instalando o Pygame no Windows". Se vir uma mensagem de erro, tente usar `pip3` no lugar de `pip`. Se nenhuma das versões estiver instalada em seu sistema, vá para a seção "Instalando o pip".

Se houver mais de uma versão de Python instalada em seu sistema, verifique se o pip está associado à versão de Python que você está usando – por exemplo, `python 3.5` em **u**. Se o pip estiver associado à versão correta de Python, vá para a seção "Instalando o Pygame no Windows". Se o pip estiver associado à versão incorreta de Python, experimente usar `pip3` no lugar de `pip`. Se nenhum dos comandos funcionar para a versão de Python que você está usando, vá para a seção "Instalando o pip".

Instalando o pip

Para instalar o pip, acesse *https://bootstrap.pypa.io/get-pip.py*. Salve o arquivo, se for solicitado. Se o código de *get-pip.py* aparecer em seu navegador, copie e cole o programa em seu editor de texto e salve o arquivo como *get-pip.py*. Depois que *get-pip.py* estiver salvo em seu computador, será necessário executá-lo com privilégios de administrador, pois o pip instalará novos pacotes em seu sistema.

NOTA Se você não conseguir encontrar *get-pip.py*, acesse *https://pip.pypa.io/*, clique em **Installation** (Instalação) no painel à esquerda e, em seguida, em "Install pip" (Instalar pip), siga o link para *get-pip.py*.

Instalando o pip no Linux e no OS X

Utilize o comando a seguir para executar *get-pip.py* com privilégios de administrador:

```
  $ sudo python get-pip.py
```

NOTA Se você usa o comando `python3` para iniciar uma sessão de terminal, utilize **`sudo python3 get-pip.py`** aqui.

Depois que o programa executar, utilize o comando **`pip --version`** (ou **`pip3 --version`**) para garantir que o pip foi instalado corretamente.

Instalando o pip no Windows

Utilize o comando a seguir para executar *get-pip.py*:

```
$ python get-pip.py
```

Se você usa um comando diferente para executar Python em um terminal, use-o para executar *get-pip.py*. Por exemplo, seu comando poderia ser `python3 get-pip.py` ou `C:\Python35\python get-pip.py`.

Depois que o programa executar, dê o comando **`python -m pip --version`** para garantir que o pip foi instalado com sucesso.

Instalando o Pygame no Linux

Se você usa Python 2.7, instale o Pygame utilizando o gerenciador de pacotes. Abra uma janela de terminal e execute o comando a seguir, que fará o download e instalará o Pygame em seu sistema:

```
$ sudo apt-get install python-pygame
```

Teste sua instalação em uma sessão de terminal com o seguinte:

```
$ python
>>> import pygame
>>>
```

Se nenhum resultado aparecer, é sinal de que Python importou o Pygame e você estará pronto para ir para a seção “Dando início ao projeto do jogo”.

Se usar Python 3, dois passos são necessários: instalar as bibliotecas das quais o Pygame depende e fazer o download e a instalação do Pygame.

Dê o comando a seguir para instalar as bibliotecas necessárias ao Pygame. (Se você usa um comando como `python3.5` em seu sistema, substitua `python3-dev` por `python3.5-dev`.)

```
$ sudo apt-get install python3-dev mercurial
```

```
$ sudo apt-get install libsdl-image1.2-dev libsdl2-dev libsdl-ttf2.0-
dev
```

Esses comandos instalarão as bibliotecas necessárias para executar a Invasão Alienígena com sucesso. Se quiser habilitar algumas funcionalidades mais sofisticadas do Pygame, por exemplo, a capacidade de adicionar sons, acrescente também as bibliotecas a seguir:

```
$ sudo apt-get install libsdl-mixer1.2-dev libportmidi-dev
$ sudo apt-get install libswscale-dev libsmpeg-dev libavformat-dev
libavcode-dev
$ sudo apt-get install python-numpy
```

Agora instale o Pygame executando o seguinte (utilize `pip3` se for apropriado ao seu sistema):

```
$ pip install --user hg+http://bitbucket.org/pygame/pygame
```

Haverá uma breve pausa na saída depois que as bibliotecas encontradas por Pygame forem apresentadas. Tecle ENTER, mesmo que algumas bibliotecas estejam faltando. Você deverá ver uma mensagem de que o Pygame foi instalado com sucesso.

Para confirmar a instalação, execute uma sessão de terminal Python e experimente importar o Pygame com o seguinte comando:

```
$ python3
>>> import pygame
>>>
```

Se isso funcionar, vá para a seção “Dando início ao projeto do jogo”.

Instalando o Pygame no OS X

Você precisará do Homebrew para instalar alguns pacotes dos quais o Pygame depende. Se você ainda não tem o Homebrew instalado, consulte o Apêndice A para ver as instruções.

Para instalar as bibliotecas das quais o Pygame depende, digite o seguinte:

```
$ brew install hg sdl sdl_image sdl_ttf
```

Esse comando instalará as bibliotecas necessárias para a execução da Invasão Alienígena. Você deverá ver uma saída rolando à medida que cada biblioteca for instalada.

Se quiser habilitar funcionalidades mais sofisticadas também, por exemplo, a inclusão de sons nos jogos, instale duas bibliotecas adicionais:

```
$ brew install sdl_mixer portmidi
```

Utilize o comando a seguir para instalar o Pygame (use `pip` no lugar de `pip3` se você usa Python 2.7):

```
$ pip3 install --user hg+http://bitbucket.org/pygame/pygame
```

Inicie uma sessão de terminal Python e importe o Pygame para verificar se a instalação foi bem-sucedida (digite `python` no lugar de `python3` se você usa Python 2.7):

```
$ python3
>>> import pygame
>>>
```

Se a instrução `import` funcionar, vá para a seção "Dando início ao projeto do jogo" a seguir.

Instalando o Pygame no Windows

O projeto Pygame está hospedado em um site de compartilhamento de código chamado Bitbucket. Para instalar o Pygame em sua versão de Windows, encontre um instalador para Windows em *https://bitbucket.org/pygame/pygame/downloads/* que corresponda à versão de Python que você utiliza. Se você não encontrar um instalador apropriado em Bitbucket, dê uma olhada em *http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame*.

Depois de baixar o arquivo apropriado, execute o instalador se for um arquivo *.exe*.

Se você tiver um arquivo terminado com *.whl*, copie esse arquivo para o diretório de seu projeto. Abra uma janela de comandos, navegue até a pasta em que você copiou o instalador e use o pip para executá-lo:

```
> python -m pip install --user pygame-1.9.2a0-cp35-none-win32.whl
```

Dando início ao projeto do jogo

Agora começaremos a desenvolver o nosso jogo, inicialmente criando uma janela vazia do Pygame na qual poderemos desenhar os elementos de nosso jogo depois, por exemplo, a espaçonave e os alienígenas. Também faremos nosso jogo responder às entradas do usuário, definiremos a cor de fundo e carregaremos a imagem de uma espaçonave.

Criando uma janela do Pygame e respondendo às entradas do usuário

Em primeiro lugar criaremos uma janela vazia do Pygame. Eis a estrutura básica de um jogo escrito com o Pygame:

alien_invasion.py

```
  import sys

  import pygame

  def run_game():
      # Inicializa o jogo e cria um objeto para a tela
u     pygame.init()
v     screen = pygame.display.set_mode((1200, 800))
      pygame.display.set_caption("Alien Invasion")

      # Inicia o laço principal do jogo
w     while True:

          # Observa eventos de teclado e de mouse
x         for event in pygame.event.get():
y             if event.type == pygame.QUIT:
                  sys.exit()

          # Deixa a tela mais recente visível
z         pygame.display.flip()

  run_game()
```

Inicialmente importamos os módulos `sys` e `pygame`. O módulo `pygame`

contém as funcionalidades necessárias para criar um jogo. Usaremos o módulo `sys` para sair do jogo quando o usuário desistir.

A Invasão Alienígena começa com a função `run_game()`. A linha `pygame.init()` em **u** inicializa as configurações de segundo plano de que o Pygame precisa para funcionar de forma apropriada. Em **v** chamamos `pygame.display.set_mode()` para criar uma janela de exibição chamada `screen`, na qual desenharemos todos os elementos gráficos do jogo. O argumento `(1200, 800)` é uma tupla que define as dimensões da janela do jogo. Ao passar essas dimensões para `pygame.display.set_mode()`, criamos uma janela de jogo com 1200 pixels de largura por 800 pixels de altura. (Esses valores podem ser ajustados de acordo com o tamanho de seu display.)

O objeto `screen` é chamado de superfície. Uma *superfície* no Pygame é uma parte da tela em que exibimos um elemento do jogo. Cada elemento do jogo, por exemplo, os alienígenas ou a espaçonave, é uma superfície. A superfície devolvida por `display.set_mode()` representa a janela inteira do jogo. Quando ativamos o laço de animação do jogo, essa superfície é automaticamente redesenhada a cada passagem pelo laço.

O jogo é controlado por um laço `while` **w** que contém um laço de eventos e o código que administra as atualizações de tela. Um *evento* é uma ação realizada pelo usuário enquanto joga, por exemplo, pressionar uma tecla ou mover o mouse. Para fazer nosso programa responder aos eventos, escreveremos um *laço de eventos* para *ouvir* um evento e executar uma tarefa apropriada de acordo com o tipo de evento ocorrido. O laço `for` em **x** é um laço de eventos.

Para acessar os eventos detectados pelo Pygame, usaremos `pygame.event.get()`. Qualquer evento de teclado ou de mouse fará o laço `for` executar. No laço, escreveremos uma série de instruções `if` para detectar e responder a eventos específicos. Por exemplo, quando o jogador clicar no botão de fechamento da janela do jogo, um evento `pygame.QUIT` será detectado e chamaremos `sys.exit()` para sair do jogo **y**.

A chamada a `pygame.display.flip()` em **z** diz ao Pygame para deixar visível a janela mais recente. Nesse caso, uma tela vazia será desenhada

sempre que passarmos pelo laço `while` para apagar a tela antiga, de modo que apenas a nova janela esteja visível. Quando movermos elementos do jogo pela tela, `pygame.display.flip()` atualizará continuamente o display para mostrar as novas posições dos elementos e ocultar as posições anteriores, criando a ilusão de um movimento suave.

A última linha nessa estrutura básica de jogo chama `run_game()`, que inicializa o jogo e o laço principal.

Execute esse código agora e você verá uma janela vazia do Pygame.

Definindo a cor de fundo

O Pygame cria uma tela preta por padrão, mas isso não é interessante. Vamos definir uma cor diferente para o plano de fundo:

alien_invasion.py

```
  --trecho omitido--
  def run_game():
      --trecho omitido--
      pygame.display.set_caption("Alien Invasion")

      # Define a cor de fundo
u     bg_color = (230, 230, 230)

      # Inicia o laço principal do jogo
      while True:

          # Observa eventos de teclado e de mouse
          --trecho omitido--

          # Redesenha a tela a cada passagem pelo laço
v         screen.fill(bg_color)

          # Deixa a tela mais recente visível
          pygame.display.flip()

  run_game()
```

Inicialmente criamos uma cor de fundo e a armazenamos em `bg_color` u.

Essa cor deve ser especificada apenas uma vez, portanto definimos seu valor antes de entrar no laço `while` principal.

As cores no Pygame são especificadas como cores RGB: uma mistura de vermelho, verde e azul. O valor de cada cor varia de 0 a 255. A cor representada pelo valor (255, 0, 0) é vermelha, por (0, 255, 0) é verde e por (0, 0, 255) é azul. Podemos misturar valores RGB para criar 16 milhões de cores. A cor cujo valor é (230, 230, 230) mistura quantidades iguais de vermelho, azul e verde, produzindo uma cor de fundo cinza-claro.

Em `v` preenchemos a tela com a cor de fundo usando o método `screen.fill()`, que aceita apenas um argumento: uma cor.

Criando uma classe de configurações

Sempre que introduzirmos uma nova funcionalidade em nosso jogo, geralmente incluiremos também algumas configurações novas. Em vez de acrescentar configurações por todo o código, vamos criar um módulo chamado `settings` que contenha uma classe de nome `Settings` para armazenar todas as configurações em um só lugar. Essa abordagem nos permite passar um objeto de configurações pelo código, em vez de passar várias configurações individuais. Além disso, ela deixa nossas chamadas de função mais simples e facilita modificar a aparência do jogo à medida que nosso projeto crescer. Para modificar o jogo, simplesmente mudaremos alguns valores em *settings.py* em vez de procurar diferentes configurações em nossos arquivos.

Aqui está a classe `Settings` inicial:

settings.py

```
class Settings():
    """Uma classe para armazenar todas as configurações da Invasão
Alienígena."""

    def __init__(self):
        """Inicializa as configurações do jogo."""
        # Configurações da tela
        self.screen_width = 1200
        self.screen_height = 800
```

```
        self.bg_color = (230, 230, 230)
```

Para criar uma instância de `Settings` e usá-la para acessar nossas configurações, modifique *alien_invasion.py* da seguinte maneira:

alien_invasion.py

```
  --trecho omitido--
  import pygame

  from settings import Settings

  def run_game():
      # Inicializa o pygame, as configurações e o objeto screen
      pygame.init()
u     ai_settings = Settings()
v     screen = pygame.display.set_mode(
          (ai_settings.screen_width, ai_settings.screen_height))
      pygame.display.set_caption("Alien Invasion")

      # Inicia o laço principal do jogo
      while True:
          --trecho omitido--
          # Redesenha a tela a cada passagem pelo laço
w         screen.fill(ai_settings.bg_color)

          # Deixa a tela mais recente visível
          pygame.display.flip()

  run_game()
```

Importamos `Settings` no arquivo principal do programa e, em seguida, criamos uma instância de `Settings` e a armazenamos em `ai_settings` depois de fazer a chamada a `pygame.init()` u. Quando criamos uma tela v, usamos os atributos `screen_width` e `screen_height` de `ai_settings` e então usamos `ai_settings` também para acessar a cor de fundo quando preenchemos a tela em w.

Adicionando a imagem de uma espaçonave

Vamos agora adicionar a espaçonave em nosso jogo. Para desenhar a espaçonave do jogador na tela, carregaremos uma imagem e usaremos o método `blit()` do Pygame para desenhá-la.

Ao escolher uma imagem artística para seus jogos, preste atenção na licença. A maneira mais segura e mais barata de começar é usar imagens com licença gratuita de um site como *http://pixabay.com/*, que possam ser modificadas.

Podemos usar praticamente qualquer tipo de arquivo de imagem no jogo, mas será mais fácil se utilizarmos um arquivo bitmap (*.bmp*) porque o Pygame carrega bitmaps por padrão. Embora possamos configurar o Pygame para usar outros tipos de arquivo, alguns desses tipos dependem de determinadas bibliotecas de imagens instaladas em seu computador. (A maioria das imagens que você encontrará estarão nos formatos *.jpg*, *.png* ou *.gif*, mas é possível convertê-las para bitmap usando ferramentas como Photoshop, GIMP e Paint.)

Em particular, preste atenção na cor de fundo das imagens escolhidas. Procure encontrar um arquivo com uma cor de fundo transparente, que possa ser substituída por qualquer cor de fundo usando um editor de imagens. Seus jogos terão melhor aparência se a cor de fundo da imagem coincidir com a cor de fundo de seu jogo. De modo alternativo, podemos fazer a cor de fundo de seu jogo coincidir com a cor de fundo da imagem.

Na Invasão Alienígena, podemos usar o arquivo *ship.bmp* (Figura 12.1), disponível nos recursos do livro em *https://www.nostarch.com/pythoncrashcourse/*. A cor de fundo do arquivo é igual às configurações usadas neste projeto. Crie uma pasta chamada *images* na pasta principal de seu projeto (*alien_invasion*). Salve o arquivo *ship.bmp* na pasta *images*.

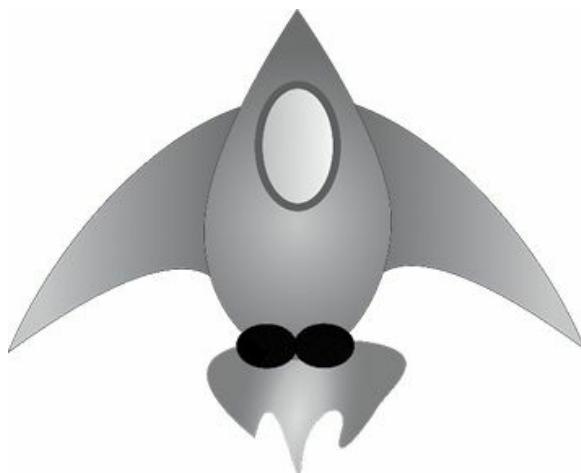



Figura 12.1 – A espaçonave da Invasão Alienígena.

Criando a classe Ship

Depois de escolher uma imagem para a espaçonave, precisamos exibi-la na tela. Para usar nossa espaçonave, criaremos um módulo chamado `ship`, que conterá a classe `Ship`. Essa classe administrará a maior parte do comportamento da espaçonave do jogador.

ship.py

```
  import pygame

  class Ship():

      def __init__(self, screen):
          """Inicializa a espaçonave e define sua posição inicial."""
          self.screen = screen

          # Carrega a imagem da espaçonave e obtém seu rect
u         self.image = pygame.image.load('images/ship.bmp')
v         self.rect = self.image.get_rect()
w         self.screen_rect = screen.get_rect()

          # Inicia cada nova espaçonave na parte inferior central da tela
x         self.rect.centerx = self.screen_rect.centerx
          self.rect.bottom = self.screen_rect.bottom

y     def blitme(self):
```

```
        """Desenha a espaçonave em sua posição atual."""
        self.screen.blit(self.image, self.rect)
```

Inicialmente importamos o módulo `pygame`. O método `__init__()` de `Ship` aceita dois parâmetros: a referência `self` e `screen`, que é a tela em que desenharemos a espaçonave. Para carregar a imagem, chamamos `pygame.image.load()` **U**. Essa função devolve uma superfície que representa a espaçonave; essa informação é armazenada em `self.image`.

Depois que a imagem é carregada, usamos `get_rect()` para acessar o atributo `rect` da superfície **V**. Um motivo para o Pygame ser tão eficiente é que ele permite tratar elementos do jogo como retângulos (*rects*), mesmo que eles não tenham exatamente o formato de um retângulo. Tratar um elemento como um retângulo é eficaz, pois os retângulos são formas geométricas simples. Essa abordagem geralmente funciona bem, a ponto de ninguém que esteja jogando perceba que não estamos trabalhando com a forma exata de cada elemento do jogo.

Quando trabalhamos com um objeto `rect`, podemos usar as coordenadas x e y das bordas superior, inferior, esquerda e direita do retângulo, assim como o centro. Podemos definir qualquer um desses valores a fim de determinar a posição atual do retângulo.

Quando um elemento do jogo for centralizado, trabalhe com os atributos `center`, `centerx` ou `centery` de um retângulo. Quando trabalhar com uma borda da tela, use os atributos `top`, `bottom`, `left` ou `right`. Quando ajustar a posição horizontal ou vertical do retângulo, você pode simplesmente usar os atributos `x` e `y`, que correspondem às coordenadas x e y de seu canto superior esquerdo. Esses atributos evitam que você precise fazer cálculos que os desenvolvedores de jogos tinham que efetuar manualmente no passado, e você perceberá que usará esses atributos com frequência.

NOTA No Pygame, a origem (0, 0) está no canto superior esquerdo da tela, e as coordenadas aumentam à medida que você descer e se deslocar para a direita. Em uma tela de 1200 por 800, a origem está no canto superior esquerdo, e o canto inferior direito tem as coordenadas (1200, 800).

Posicionaremos a espaçonave na parte inferior central da tela. Para isso, inicialmente armazene o retângulo da tela em `self.screen_rect` **W** e, em seguida, faça o valor de `self.rect.centerx` (a coordenada x do centro da

espaçonave) coincidir com o atributo `centerx` do retângulo da tela x. Faça com que o valor de `self.rect.bottom` (a coordenada y da parte inferior da espaçonave) seja igual ao valor do atributo `bottom` do retângulo da tela. O Pygame usará esses atributos `rect` para posicionar a imagem da espaçonave de modo que ela esteja centralizada horizontalmente e alinhada com a parte inferior da tela.

Em y definimos o método `blitme()`, que desenhará a imagem na tela na posição especificada por `self.rect`.

Desenhando a espaçonave na tela

Vamos agora atualizar *alien_invasion.py* para criar uma espaçonave e chamar o seu método `blitme()`:

alien_invasion.py

```
  --trecho omitido--
  from settings import Settings
  from ship import Ship

  def run_game():
      --trecho omitido--
      pygame.display.set_caption("Alien Invasion")

      # Cria uma espaçonave
u     ship = Ship(screen)

      # Inicia o laço principal do jogo
      while True:
          --trecho omitido--
          # Redesenha a tela a cada passagem pelo laço
          screen.fill(ai_settings.bg_color)
v         ship.blitme()

          # Deixa a tela mais recente visível
          pygame.display.flip()

  run_game()
```

Importamos `Ship` e então criamos uma instância de `Ship` (chamada `ship`)

depois que a tela foi criada. Essa operação deve estar antes do laço `while` principal **u** para que uma nova instância da espaçonave não seja criada a cada passagem pelo laço. Desenhamos a espaçonave na tela chamando `ship.blitme()` depois de preencher a cor de fundo; assim a espaçonave aparecerá sobre essa cor **v**.

Se executarmos *alien_invasion.py* agora, veremos uma tela de jogo vazia, com nossa espaçonave parada na parte inferior central, como mostra a Figura 12.2.

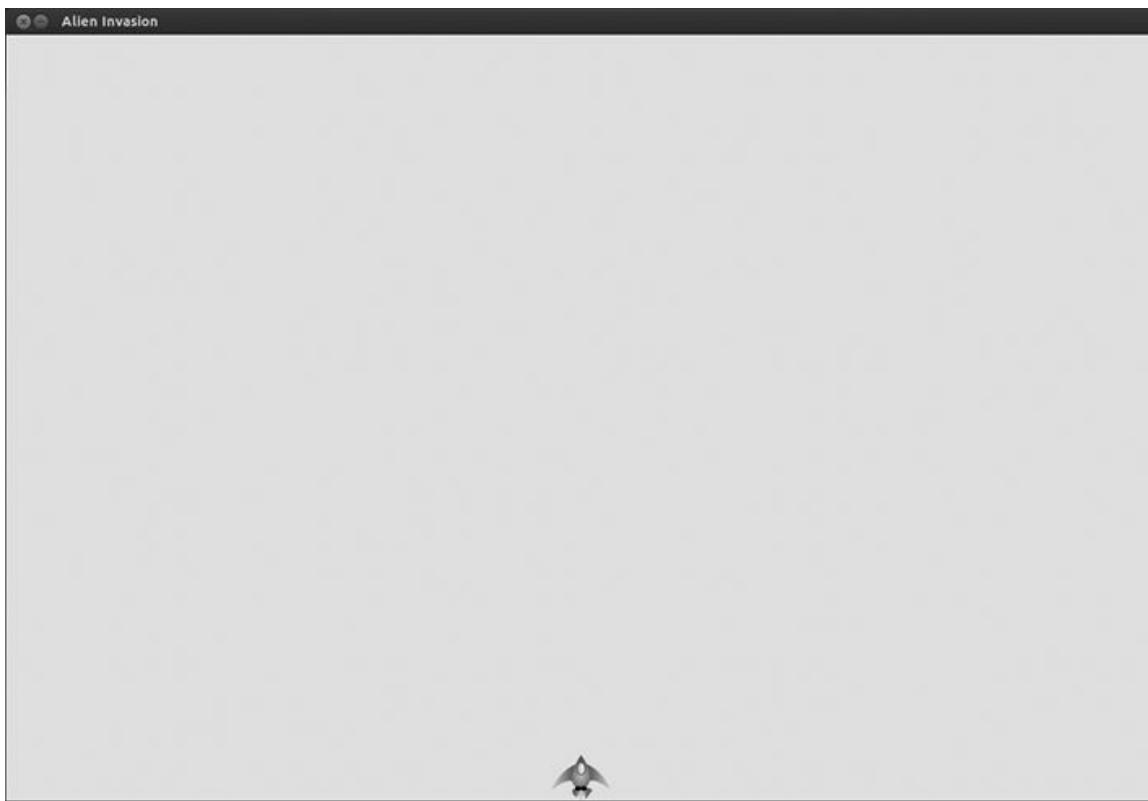



Figura 12.2 – A Invasão Alienígena com a espaçonave na parte inferior central da tela.

Refatoração: o módulo game_functions

Em projetos maiores, com frequência, você vai refatorar códigos já escritos antes de adicionar novos códigos. A *refatoração* simplifica a estrutura do código que você já escreveu, facilitando expandi-lo. Nesta seção criaremos um novo módulo chamado `game_functions`, no qual armazenaremos várias funções que farão a Invasão Alienígena executar. O módulo `game_functions` evitará que *alien_invasion.py* fique longo demais e deixará sua lógica mais

fácil de compreender.

Função check_events()

Começaremos transferindo o código que administra eventos para uma função separada chamada `check_events()`. Isso simplificará `run_game()` e isolará o laço de gerenciamento de eventos. Isolar o laço de eventos permite administrar os eventos de forma separada de outros aspectos do jogo, por exemplo, da atualização da tela.

Coloque `check_events()` em um módulo separado chamado `game_functions`:

game_functions.py

```
import sys

import pygame

def check_events():
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Esse módulo importa `sys` e `pygame`, usados no laço de verificação de eventos. A função não precisa de nenhum parâmetro no momento, e seu corpo foi copiado do laço de eventos em *alien_invasion.py*.

Vamos agora modificar *alien_invasion.py* para que importe o módulo `game_functions`, e substituiremos o laço de eventos por uma chamada a `check_events()`:

alien_invasion.py

```
import pygame

from settings import Settings
from ship import Ship
import game_functions as gf

def run_game():
```

```
    --trecho omitido--
    # Inicia o laço principal do jogo
    while True:
        gf.check_events()

        # Redesenha a tela a cada passagem pelo laço
        --trecho omitido--
```

Não precisamos mais importar `sys` diretamente no arquivo principal do programa, pois ele é usado apenas no módulo `game_functions` agora. Atribuímos o alias `gf` ao módulo importado `game_functions` para simplificar.

Função update_screen()

Vamos transferir o código de atualização da tela para uma função separada de nome `update_screen()` em *game_functions.py* para simplificar mais a função `run_game()`:

game_functions.py

```
--trecho omitido--

def check_events():
    --trecho omitido--

def update_screen(ai_settings, screen, ship):
    """Atualiza as imagens na tela e alterna para a nova tela."""
    # Redesenha a tela a cada passagem pelo laço
    screen.fill(ai_settings.bg_color)
    ship.blitme()

    # Deixa a tela mais recente visível
    pygame.display.flip()
```

A nova função `update_screen()` aceita três parâmetros: `ai_settings`, `screen` e `ship`. Agora precisamos atualizar o laço `while` de *alien_invasion.py* com uma chamada a `update_screen()`:

alien_invasion.py

```
--trecho omitido--
```

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events()
        gf.update_screen(ai_settings, screen, ship)

run_game()
```

Essas duas funções deixam o laço `while` mais simples e facilitará os desenvolvimentos futuros. Em vez de trabalhar em `run_game()`, podemos fazer a maior parte de nossas tarefas no módulo `game_functions`.

Como queríamos começar a trabalhar com o código em um único arquivo, não havíamos introduzido o módulo `game_functions` de imediato. Essa abordagem dá uma ideia de um processo realista de desenvolvimento: comece escrevendo seu código do modo mais simples possível e refatore à medida que seu projeto se tornar mais complexo.

Agora que o nosso código está reestruturado de modo a facilitar a adição de novos códigos, podemos trabalhar com os aspectos dinâmicos do jogo!

FAÇA VOCÊ MESMO

12.1 – Céu azul: Crie uma janela do Pygame com uma cor de fundo azul.

12.2 – Personagem do jogo: Encontre uma imagem de bitmap de um personagem de jogo que você goste ou converta uma imagem em um bitmap. Crie uma classe que desenhe o personagem no centro da tela e faça a cor de fundo da imagem coincidir com a cor de fundo da tela ou vice-versa.

Pilotando a espaçonave

Vamos dar ao jogador a capacidade de mover a espaçonave para a direita e para a esquerda. Para isso, escreveremos um código que responda quando o jogador pressionar a seta para a direita ou para a esquerda. Vamos nos concentrar no movimento para a direita antes e então aplicaremos os mesmos princípios para controlar o movimento para a esquerda. Ao fazer isso, você aprenderá a controlar o movimento das imagens na tela.

Respondendo a um pressionamento de tecla

Sempre que o jogador pressionar uma tecla, esse pressionamento será registrado no Pygame como um evento. Todo evento é capturado pelo método `pygame.event.get()`, portanto precisamos especificar quais são os tipos de eventos que queremos verificar em nossa função `check_events()`. Todo pressionamento de tecla é registrado como um evento `KEYDOWN`.

Quando um evento `KEYDOWN` é detectado, devemos verificar se a tecla pressionada dispara um determinado evento. Por exemplo, se a seta para a direita for pressionada, aumentamos o valor de `rect.centerx` da espaçonave para movê-la para a direita:

game_functions.py

```
  def check_events(ship):
      """Responde a eventos de pressionamento de teclas e de mouse."""
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              sys.exit()

u         elif event.type == pygame.KEYDOWN:
v             if event.key == pygame.K_RIGHT:
                  # Move a espaçonave para a direita
w                 ship.rect.centerx += 1
```

Fornecemos um parâmetro `ship` à função `check_events()` porque a nave deve se deslocar para a direita quando a tecla para a direita for pressionada. Em `check_events()`, acrescentamos um bloco `elif` no laço de eventos para responder quando o Pygame detectar um evento `KEYDOWN` **u**. Verificamos se a tecla pressionada é a seta para a direita (`pygame.K_RIGHT`) lendo o atributo `event.key` **v**. Se a seta para a direita foi pressionada, movemos a espaçonave para a direita incrementando o valor de `ship.rect.centerx` de 1 **w**.

Precisamos atualizar a chamada a `check_events()` em *alien_invasion.py* para que `ship` seja passado como argumento:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ship)
        gf.update_screen(ai_settings, screen, ship)
```

Se *alien_invasion.py* for executado agora, devemos ver a nave mover-se para a direita em um pixel sempre que a seta para a direita for pressionada. É um começo, mas não é um modo eficiente de controlar a espaçonave. Vamos melhorar esse controle para permitir um movimento contínuo.

Permitindo um movimento contínuo

Quando o jogador manter a seta para a direita pressionada, queremos que a espaçonave continue a se mover para a direita até o jogador soltar a tecla. Faremos nosso jogo detectar um evento `pygame.KEYUP` para que possamos saber quando a seta para a direita foi solta; então usaremos os eventos `KEYDOWN` e `KEYUP` juntamente com uma flag chamada `moving_right` para implementar o movimento contínuo.

Quando a espaçonave estiver parada, a flag `moving_right` será `False`. Quando a seta para a direita for pressionada, definiremos a flag com `True`; quando essa tecla for solta, definiremos a flag com `False` novamente.

A classe `Ship` controla todos os atributos da espaçonave, portanto lhe daremos um atributo chamado `moving_right` e um método `update()` para verificar o status da flag `moving_right`. O método `update()` mudará a posição da espaçonave se a flag estiver definida com `True`. Chamaremos esse método sempre que quisermos atualizar a posição da espaçonave.

Eis as mudanças feitas na classe `Ship`:

ship.py

```
  class Ship():

      def __init__(self, screen):
          --trecho omitido--
          # Inicia cada nova espaçonave na parte inferior central da tela
          self.rect.centerx = self.screen_rect.centerx
          self.rect.bottom = self.screen_rect.bottom

          # Flag de movimento
u         self.moving_right = False

v     def update(self):
```

```
        """Atualiza a posição da espaçonave de acordo com a flag de
movimento."""
        if self.moving_right:
            self.rect.centerx += 1

    def blitme(self):
        --trecho omitido--
```

Adicionamos um atributo `self.moving_right` no método `__init__()` e o definimos com `False` inicialmente **u**. Então acrescentamos `update()`, que move a espaçonave para a direita se a flag for `True` **v**.

Agora modifique `check_events()` para que `moving_right` seja definido com `True` quando a seta para a direita for pressionada e com `False` quando essa tecla for solta:

game_functions.py

```
def check_events(ship):
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        --trecho omitido--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
u               ship.moving_right = True

v       elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = False
```

Em **u** modificamos o modo como o jogo responde quando o jogador pressiona a seta para a direita; em vez de mudar a posição da espaçonave de forma direta, simplesmente definimos `moving_right` com `True`. Em **v** adicionamos um novo bloco `elif` que responde a eventos `KEYUP`. Quando o jogador soltar a tecla de seta para a direita (`K_RIGHT`), definiremos `moving_right` com `False`.

Por fim, modificamos o laço `while` em *alien_invasion.py* para que o método `update()` da espaçonave seja chamado a cada passagem pelo laço:

alien_invasion.py

```
# Inicia o laço principal do jogo
while True:
    gf.check_events(ship)
    ship.update()
    gf.update_screen(ai_settings, screen, ship)
```

A posição da espaçonave será atualizada depois que verificarmos os eventos de teclado e antes de atualizarmos a tela. Isso permite que a posição da nave seja atualizada em resposta a uma entrada do jogador e garante que a posição atualizada seja usada quando a espaçonave for desenhada na tela.

Ao executar *alien_invasion.py* e manter a seta para a direita pressionada, a espaçonave deverá mover-se continuamente para a direita até que a tecla seja solta.

Movendo tanto para a esquerda quanto para a direita

Agora que a espaçonave é capaz de se mover continuamente para a direita, adicionar um movimento para a esquerda é fácil. Modificaremos novamente a classe `Ship` e a função `check_events()`. A seguir, apresentamos as alterações relevantes em `__init__()` e em `update()` na classe `Ship`:

ship.py

```
def __init__(self, screen):
    --trecho omitido--
    # Flags de movimento
    self.moving_right = False
    self.moving_left = False

def update(self):
    """Atualiza a posição da espaçonave de acordo com as flags de
movimento."""
    if self.moving_right:
        self.rect.centerx += 1
    if self.moving_left:
        self.rect.centerx -= 1
```

Em `__init__()`, acrescentamos uma flag `self.moving_left`. Em `update()` usamos dois blocos `if` separados em vez de utilizar um `elif` em `update()`

para permitir que o valor `rect.centerx` da espaçonave seja incrementado e então decrementado se as duas teclas de direção forem mantidas pressionadas. Isso resulta na espaçonave parada. Se usássemos `elif` para o movimento à esquerda, a seta para a direita sempre teria prioridade. Fazer isso dessa maneira deixa os movimentos mais precisos ao alternamos o movimento da esquerda para a direita, quando o jogador poderia momentaneamente manter as duas teclas pressionadas.

Precisamos fazer dois ajustes em `check_events()`:

game_functions.py

```
def check_events(ship):
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        --trecho omitido--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                ship.moving_left = False
```

Se um evento `KEYDOWN` ocorrer para a tecla `K_LEFT`, definimos `moving_left` com `True`. Se um evento `KEYUP` ocorrer para a tecla `K_LEFT`, definimos `moving_left` com `False`. Podemos usar blocos `elif` nesse caso, pois cada evento está associado a apenas uma tecla. Se o jogador pressionar as duas teclas ao mesmo tempo, dois eventos diferentes serão detectados.

Se *alien_invasion.py* for executado agora, você deverá ser capaz de mover a espaçonave continuamente para a direita e para a esquerda. Se mantiver as duas teclas pressionadas, a espaçonave deverá parar de se mover.

Em seguida, aperfeiçoaremos o movimento da espaçonave. Vamos ajustar a velocidade dela e limitar a distância que a espaçonave pode percorrer para que ela não desapareça nas laterais da tela.

Ajustando a velocidade da espaçonave

No momento, a espaçonave se desloca de um pixel por ciclo do laço `while`, mas podemos ter um controle mais minucioso da velocidade da espaçonave acrescentando um atributo `ship_speed_factor` na classe `Settings`. Usaremos esse atributo para determinar a distância com que a espaçonave se deslocará a cada passagem pelo laço. Eis o novo atributo em *settings.py*:

settings.py

```
class Settings():
    """Uma classe para armazenar todas as configurações da Invasão
Alienígena."""

    def __init__(self):
        --trecho omitido--

        # Configurações da espaçonave
        self.ship_speed_factor = 1.5
```

Definimos o valor inicial de `ship_speed_factor` com `1.5`. Quando quisermos mover a espaçonave, ajustaremos sua posição em 1,5 pixel, e não em 1 pixel.

Usamos valores decimais para a configuração da velocidade para que possamos ter um controle mais preciso da velocidade da espaçonave quando aumentarmos o ritmo do jogo mais tarde. No entanto, os atributos de retângulo como `centerx` armazenam apenas valores inteiros, portanto precisamos fazer algumas modificações em `Ship`:

ship.py

```
class Ship():

u     def __init__(self, ai_settings, screen):
          """Inicializa a espaçonave e define sua posição inicial."""
          self.screen = screen
v         self.ai_settings = ai_settings
          --trecho omitido--
```

```
        # Inicia cada nova espaçonave na parte inferior central da tela
        --trecho omitido--

        # Armazena um valor decimal para o centro da espaçonave
w        self.center = float(self.rect.centerx)

        # Flags de movimento
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Atualiza a posição da espaçonave de acordo com as flags de
movimento."""
        # Atualiza o valor do centro da espaçonave, e não o retângulo
        if self.moving_right:
x            self.center += self.ai_settings.ship_speed_factor
        if self.moving_left:
            self.center -= self.ai_settings.ship_speed_factor

        # Atualiza o objeto rect de acordo com self.center
y        self.rect.centerx = self.center

    def blitme(self):
        --trecho omitido--
```

Em u adicionamos `ai_settings` à lista de parâmetros de `__init__()` para que a espaçonave tenha acesso à sua configuração de velocidade. Então transformamos o parâmetro `ai_settings` em um atributo para que possamos usá-lo em `update()` v. Agora que estamos ajustando a posição da espaçonave em frações de um pixel, precisamos armazenar a posição em uma variável capaz de armazenar um valor decimal. Você pode usar um valor decimal para definir um atributo de `rect`, mas `rect` armazenará apenas a parte inteira desse valor. Para armazenar a posição da espaçonave de forma precisa, definimos um novo atributo `self.center`, capaz de armazenar valores decimais w. Usamos a função `float()` para converter o valor de `self.rect.centerx` em um decimal e armazenamos esse valor em `self.center`.

Agora, quando alterarmos a posição da espaçonave em `update()`, o valor

de `self.center` será ajustado de acordo com a quantidade armazenada em `ai_settings.ship_speed_factor` x. Depois que `self.center` é atualizado, usamos o novo valor para atualizar `self.rect.centerx`, que controla a posição da espaçonave y. Somente a parte inteira de `self.center` será armazenada em `self.rect.centerx`, mas isso não é um problema para exibir a espaçonave.

Devemos passar `ai_settings` como argumento quando criarmos uma instância de `Ship` em *alien_invasion.py*:

alien_invasion.py

```
--trecho omitido--
def run_game():
    --trecho omitido--
    # Cria uma espaçonave
    ship = Ship(ai_settings, screen)
    --trecho omitido--
```

Agora qualquer valor de `ship_speed_factor` maior que um fará a espaçonave se deslocar mais rapidamente. Isso será útil para fazer com que a espaçonave responda rápido o suficiente para atingir os alienígenas, e nos permitirá mudar o ritmo do jogo à medida que o jogador fizer progressos no gameplay.

Limitando o alcance da espaçonave

A essa altura, a espaçonave desaparecerá nas bordas da tela se você mantiver a tecla de direção pressionada por tempo suficiente. Vamos corrigir isso de modo que a espaçonave pare de se mover quando alcançar a borda da tela. Fazemos isso modificando o método `update()` em `Ship`:

ship.py

```
    def update(self):
        """Atualiza a posição da espaçonave de acordo com as flags de
 movimento."""
        # Atualiza o valor do centro da espaçonave, e não o retângulo
u        if self.moving_right and self.rect.right <
self.screen_rect.right:
```

```
            self.center += self.ai_settings.ship_speed_factor
v       if self.moving_left and self.rect.left > 0:
            self.center -= self.ai_settings.ship_speed_factor

        # Atualiza o objeto rect de acordo com self.center
        self.rect.centerx = self.center
```

Esse código verifica a posição da espaçonave antes de alterar o valor de `self.center`. O código `self.rect.right` devolve o valor da coordenada x da borda direita do rect da espaçonave. Se esse valor for menor que o valor devolvido por `self.screen_rect.right`, é sinal de que a espaçonave não alcançou a borda direita da tela **u**. O mesmo vale para a borda esquerda: se o valor do lado esquerdo de `rect` for maior que zero, a espaçonave não atingiu a borda esquerda da tela **v**. Isso garante que a espaçonave esteja dentro desses limites antes de ajustar o valor de `self.center`.

Se você executar *alien_invasion.py* agora, a espaçonave interromperá o movimento em qualquer borda da tela.

Refatorando check_events()

A função `check_events()` aumentará de tamanho à medida que continuarmos a desenvolver o jogo, portanto vamos dividi-la em duas funções diferentes: uma que trate eventos `KEYDOWN` e outra para tratar eventos `KEYUP`:

game_functions.py

```
def check_keydown_events(event, ship):
    """Responde a pressionamentos de tecla."""
    if event.key == pygame.K_RIGHT:
        ship.moving_right = True
    elif event.key == pygame.K_LEFT:
        ship.moving_left = True

def check_keyup_events(event, ship):
    """Responde a solturas de tecla."""
    if event.key == pygame.K_RIGHT:
        ship.moving_right = False
    elif event.key == pygame.K_LEFT:
```

```
        ship.moving_left = False

def check_events(ship):
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            check_keydown_events(event, ship)
        elif event.type == pygame.KEYUP:
            check_keyup_events(event, ship)
```

Criamos duas novas funções: `check_keydown_events()` e `check_keyup_events()`. Cada uma delas precisa de um parâmetro `event` e de um parâmetro `ship`. Os corpos dessas duas funções foram copiados de `check_events()` e substituímos o código antigo por chamadas às novas funções. A função `check_events()` está mais simples agora, com uma estrutura de código mais limpa, o que facilitará o desenvolvimento de outras respostas a entradas do usuário.

Uma recapitulação rápida

Na próxima seção, acrescentaremos a capacidade de atirar, o que envolve um novo arquivo chamado *bullet.py* e modificações em alguns dos arquivos que já temos. Neste momento, temos quatro arquivos que contêm diversas classes, funções e métodos. Para deixar claro o modo como o nosso projeto está organizado, vamos rever cada um desses arquivos antes de acrescentar outras funcionalidades.

alien_invasion.py

O arquivo principal *alien_invasion.py* cria vários objetos importantes usados no jogo: as configurações são armazenadas em `ai_settings`, a superfície principal de display é armazenada em `screen` e uma instância de `ship` é criada nesse arquivo. Também em *alien_invasion.py* está o laço principal do jogo: um laço `while` que chama `check_events()`, `ship.update()` e `update_screen()`.

alien_invasion.py é o único arquivo que deve ser executado quando você quiser jogar Invasão Alienígena. Os outros arquivos – *settings.py*, *game_functions.py*, *ship.py* – contêm códigos que são importados, de forma direta ou não, nesse arquivo.

settings.py

O arquivo *settings.py* contém a classe `Settings`. Essa classe tem apenas um método `__init__()`, que inicializa os atributos para controlar a aparência do jogo e a velocidade da espaçonave.

game_functions.py

O arquivo *game_functions.py* contém várias funções que executam a maior parte das tarefas do jogo. A função `check_events()` detecta eventos relevantes, como pressionamentos e solturas de teclas, além de processar cada um desses tipos de evento por meio das funções auxiliares `check_keydown_events()` e `check_keyup_events()`. Por enquanto, essas funções administram o movimento da espaçonave. O módulo `game_functions` também contém `update_screen()`, que redesenha a tela a cada passagem pelo laço principal.

ship.py

O arquivo *ship.py* contém a classe `Ship`. A classe `Ship` tem um método `__init__()`, um método `update()` para administrar a posição da espaçonave e um método `blitme()` para desenhar a espaçonave na tela. A imagem propriamente dita da espaçonave está armazenada em *ship.bmp*, que está na pasta *images*.

FAÇA VOCÊ MESMO

12.3 – Foguete: Crie um jogo que comece com um foguete no centro da tela. Permita que o jogador mova o foguete para cima, para baixo, para a direita e para a esquerda usando as quatro teclas de direção. Garanta que o foguete não se desloque para além de qualquer borda da tela.

12.4 – Teclas: Em um arquivo Pygame, crie uma tela vazia. No laço de eventos,

exiba o atributo `event.key` sempre que o evento `pygame.KEYDOWN` for detectado. Execute o programa e pressione várias teclas para ver como o Pygame responde.

Atirando

Vamos agora acrescentar a capacidade de atirar. Escreveremos um código que lance um projétil (um pequeno retângulo) quando o jogador pressionar a barra de espaço. Os projéteis se deslocarão para cima na tela até desaparecerem ao ultrapassar a parte superior da tela.

Adicionando as configurações dos projéteis

Em primeiro lugar, atualize *settings.py* para incluir os valores de que precisaremos para uma nova classe `Bullet` no final do método `__init__()`:

settings.py

```
    def __init__(self):
        --trecho omitido--
        # Configurações dos projéteis
        self.bullet_speed_factor = 1
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
```

Essas configurações criam projéteis cinza-escuros, com largura de 3 pixels e altura de 15 pixels. Os projéteis se deslocarão de modo um pouco mais lento que a espaçonave.

Criando a classe Bullet

Agora crie um arquivo *bullet.py* para armazenar nossa classe `Bullet`. Eis a primeira parte de *bullet.py*:

bullet.py

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
```

```
    """Uma classe que administra projéteis disparados pela
espaçonave"""

    def __init__(self, ai_settings, screen, ship):
        """Cria um objeto para o projétil na posição atual da
espaçonave."""
        super(Bullet, self).__init__()
        self.screen = screen

        # Cria um retângulo para o projétil em (0, 0) e, em seguida,
define a
        # posição correta
u       self.rect = pygame.Rect(0, 0, ai_settings.bullet_width,
            ai_settings.bullet_height)
v       self.rect.centerx = ship.rect.centerx
w       self.rect.top = ship.rect.top

        # Armazena a posição do projétil como um valor decimal
x       self.y = float(self.rect.y)

y       self.color = ai_settings.bullet_color
        self.speed_factor = ai_settings.bullet_speed_factor
```

A classe `Bullet` herda de `Sprite`, que importamos do módulo `pygame.sprite`. Ao usar sprites, podemos agrupar elementos relacionados no jogo e atuar em todos os elementos agrupados de uma só vez. Para criar uma instância de um projétil, `__init__()` precisa das instâncias `ai_settings`, `screen` e `ship`; além disso, chamamos `super()` para herdar de modo apropriado de `Sprite`.

NOTA A chamada a `super(Bullet, self).__init__()` usa a sintaxe de Python 2.7. Essa sintaxe funciona em Python 3 também; de modo alternativo, essa chamada pode ser feita de forma mais simples como `super().__init__()`.

Em **u** criamos o atributo `rect` do projétil. O projétil não está baseado em uma imagem, portanto precisamos criar um retângulo do zero usando a classe `pygame.Rect()`. Essa classe exige as coordenadas x e y do canto superior esquerdo do `rect`, além de sua largura e sua altura. Inicializamos o `rect` em (0, 0), mas ele será movido para o local correto nas duas próximas linhas porque a posição do projétil depende da posição da espaçonave. Os

valores da largura e da altura do projétil são obtidos dos valores armazenados em ai_settings.

Em v definimos o centerx do projétil para que seja igual ao rect.centerx da espaçonave. O projétil deve emergir da parte superior da espaçonave, portanto definimos o rect do projétil para que sua parte superior coincida com a parte superior do rect da espaçonave, fazendo parecer que o projétil foi disparado da espaçonave w.

Armazenamos um valor decimal para a coordenada y do projétil para que possamos fazer ajustes mais precisos em sua velocidade x. Em y armazenamos a cor e as configurações de velocidade do projétil em self.color e em self.speed_factor.

A seguir, apresentamos a segunda parte de *bullet.py*, que contém update() e draw_bullet():

bullet.py

```
     def update(self):
         """Move o projétil para cima na tela."""
         # Atualiza a posição decimal do projétil
u        self.y -= self.speed_factor
         # Atualiza a posição de rect
v        self.rect.y = self.y

     def draw_bullet(self):
         """Desenha o projétil na tela."""
w        pygame.draw.rect(self.screen, self.color, self.rect)
```

O método update() administra a posição do projétil. Quando um projétil é disparado, ele se move para cima na tela, o que corresponde a um decréscimo no valor da coordenada y; portanto, para atualizar a posição, subtraímos de self.y a quantidade armazenada em self.speed_factor u. Então usamos o valor de self.y para definir o valor de self.rect.y v. O atributo speed_factor nos permite aumentar a velocidade dos projéteis à medida que o jogo progredir ou conforme for necessário para melhor ajustar o comportamento do jogo. Depois que o projétil é disparado, o valor de sua coordenada x não muda, portanto ele só se deslocará na vertical, em linha reta.

Se quisermos desenhar um projétil, chamaremos `draw_bullet()`. A função `draw.rect()` preenche a parte da tela definida pelo `rect` do projétil com a cor armazenada em `self.color` w.

Armazenando projéteis em um grupo

Agora que temos uma classe `Bullet` e as configurações necessárias definidas, podemos escrever um código para disparar um projétil sempre que o jogador pressionar a barra de espaço. Inicialmente criaremos um grupo em *alien_invasion.py* para armazenar todos os projéteis ativos; desse modo podemos administrar todos os projéteis que tenham sido disparados. Esse grupo será uma instância da classe `pygame.sprite.Group`, que se comporta como uma lista; essa classe contém algumas funcionalidades extras que são úteis no desenvolvimento de jogos. Usaremos esse grupo para desenhar os projéteis na tela a cada passagem pelo laço principal e atualizar a posição de cada projétil:

alien_invasion.py

```
  import pygame
  from pygame.sprite import Group
  --trecho omitido--

  def run_game():
      --trecho omitido--
      # Cria uma espaçonave
      ship = Ship(ai_settings, screen)
      # Cria um grupo no qual serão armazenados os projéteis
u     bullets = Group()

      # Inicia o laço principal do jogo
      while True:
          gf.check_events(ai_settings, screen, ship, bullets)
          ship.update()
v         bullets.update()
          gf.update_screen(ai_settings, screen, ship, bullets)

  run_game()
```

Importamos `Group` de `pygame.sprite`. Em **u** criamos uma instância de `Group` e a chamamos de `bullets`. Esse grupo é criado fora do laço `while` para que um novo grupo de projéteis não seja criado a cada ciclo do laço.

NOTA Se você criar um grupo como esse dentro do laço, estará criando milhares de grupos de projéteis, e seu jogo provavelmente ficará muito lento. Se seu jogo travar, observe com atenção o que está acontecendo em seu laço `while` principal.

Passamos `bullets` para `check_events()` e para `update_screen()`. Teremos que trabalhar com `bullets` em `check_events()` quando a barra de espaço for pressionada, e precisaremos atualizar os projéteis desenhados na tela em `update_screen()`.

Quando chamamos `update()` em um grupo **v**, ele chamará `update()` automaticamente para cada sprite do grupo. A linha `bullets.update()` chama `bullet.update()` para cada projétil que colocamos no grupo `bullets`.

Disparando os projéteis

Em *game_functions.py* precisamos modificar `check_keydown_events()` para disparar um projétil quando a barra de espaço for pressionada. Não precisamos mudar `check_keyup_events()` porque nada acontece quando a tecla é solta. Também devemos modificar `update_screen()` para garantir que cada projétil seja redesenhado na tela antes de `flip()` ser chamada. Eis as alterações relevantes feitas em *game_functions.py*:

game_functions.py

```
  --trecho omitido--
  from bullet import Bullet

u def check_keydown_events(event, ai_settings, screen, ship, bullets):
      --trecho omitido--
v     elif event.key == pygame.K_SPACE:
          # Cria um novo projétil e o adiciona ao grupo de projéteis
          new_bullet = Bullet(ai_settings, screen, ship)
          bullets.add(new_bullet)
         --trecho omitido--

w def check_events(ai_settings, screen, ship, bullets):
```

```
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        --trecho omitido--
        elif event.type == pygame.KEYDOWN:
            check_keydown_events(event, ai_settings, screen, ship,
  bullets)
        --trecho omitido--

x def update_screen(ai_settings, screen, ship, bullets):
    --trecho omitido--
    # Redesenha todos os projéteis atrás da espaçonave e dos
  alienígenas
y   for bullet in bullets.sprites():
        bullet.draw_bullet()
    ship.blitme()
    --trecho omitido--
```

O grupo `bullets` é passado para `check_keydown_events()` u. Quando o jogador pressiona a barra de espaço, criamos um novo projétil (uma instância de `Bullet` que chamamos de `new_bullet`) e o adicionamos ao grupo `bullets` v usando o método `add()`; o código `bullets.add(new_bullet)` armazena o novo projétil no grupo `bullets`.

Precisamos acrescentar `bullets` como parâmetro na definição de `check_events()` w e passar `bullets` como argumento na chamada a `check_keydown_events()` também.

Passamos o parâmetro `bullets` para `update_screen()` em x, que desenha os projéteis na tela. O método `bullets.sprites()` devolve uma lista de todos os sprites do grupo `bullets`. Para desenhar todos os projéteis disparados na tela, percorremos os sprites em `bullets` com um laço e chamamos `draw_bullet()` em cada um y.

Se *alien_invasion.py* for executado agora, você deverá ser capaz de mover a espaçonave para a direita e para a esquerda e disparar quantos projéteis quiser. Os projéteis se deslocarão para cima na tela e desaparecerão quando alcançarem a parte superior, como mostra a Figura 12.3. Você pode alterar o tamanho, a cor e a velocidade dos projéteis em *settings.py*.

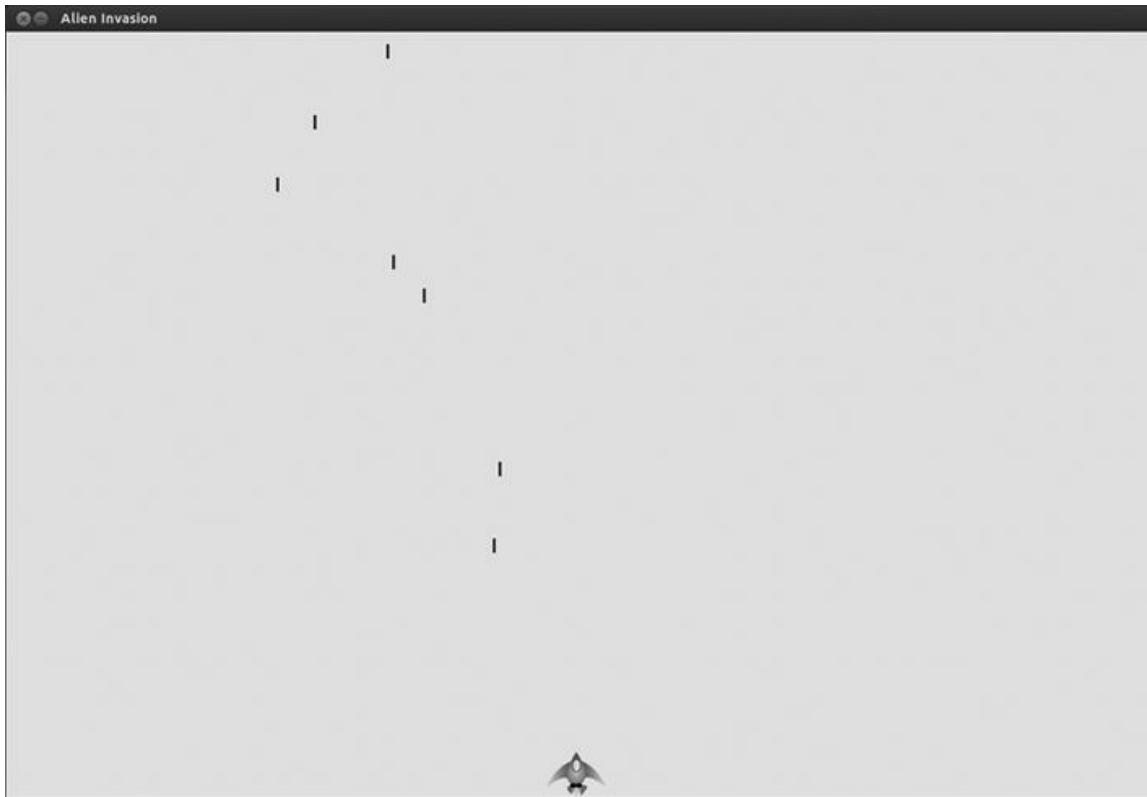



Figura 12.3 – A espaçonave após o disparo de uma série de projéteis.

Apagando projéteis antigos

No momento, os projéteis desaparecem quando alcançam a parte superior, mas somente porque o Pygame não é capaz de desenhá-los acima da parte superior da tela. Na verdade, os projéteis continuam existindo; os valores de suas coordenadas y simplesmente assumem valores negativos cada vez menores. Isso é um problema porque eles continuam consumindo memória e capacidade de processamento.

Precisamos nos livrar desses projéteis antigos, ou o jogo ficará lento por executar tantas tarefas desnecessárias. Para isso, precisamos detectar se o valor `bottom` do `rect` de um projétil tem um valor igual a 0, o que indica que o projétil ultrapassou a parte superior da tela:

alien_invasion.py

```
# Inicia o laço principal do jogo
while True:
    gf.check_events(ai_settings, screen, ship, bullets)
```

```
        ship.update()
        bullets.update()

        # Livra-se dos projéteis que desapareceram
u       for bullet in bullets.copy():
v           if bullet.rect.bottom <= 0:
w               bullets.remove(bullet)
x       print(len(bullets))

        gf.update_screen(ai_settings, screen, ship, bullets)
```

Não devemos remover itens de uma lista ou de um grupo em um laço `for`, portanto precisamos usar uma cópia do grupo no laço. Utilizamos o método `copy()` para preparar o laço `for` u, o que nos permite modificar `bullets` no laço. Verificamos cada projétil para ver se ele desapareceu por ter ultrapassado a parte superior da tela em v. Em caso afirmativo, o projétil é removido de `bullets` w. Em x inserimos uma instrução `print` para mostrar quantos projéteis existem no momento no jogo e conferir se estão sendo apagados.

Se esse código funcionar corretamente, poderemos observar a saída no terminal enquanto disparamos os projéteis e ver que o número de projéteis se reduz a zero depois que cada conjunto de projéteis desaparece na parte superior da tela. Depois de executar o jogo e confirmar que os projéteis são devidamente apagados, remova a instrução `print`. Se você deixar essa instrução, o jogo ficará significativamente mais lento, pois escrever uma saída no terminal demora mais que desenhar imagens na janela do jogo.

Limitando o número de projéteis

Muitos jogos em que há disparos limitam o número de projéteis que um jogador pode ter na tela ao mesmo tempo para incentivar os jogadores a atirarem de forma precisa. Faremos o mesmo na Invasão Alienígena.

Inicialmente armazene o número de projéteis permitidos em *settings.py*:

settings.py

```
        # Configurações dos projéteis
        self.bullet_width = 3
```

```
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3
```

Essa instrução limita o jogador a três projéteis ao mesmo tempo. Usaremos essa configuração em *game_functions.py* para verificar quantos projéteis existem antes de criar um novo projétil em `check_keydown_events()`:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --trecho omitido--
    elif event.key == pygame.K_SPACE:
        # Cria um novo projétil e o adiciona ao grupo de projéteis
        if len(bullets) < ai_settings.bullets_allowed:
            new_bullet = Bullet(ai_settings, screen, ship)
            bullets.add(new_bullet)
```

Quando a barra de espaço é pressionada, verificamos o tamanho de `bullets`. Se `len(bullets)` for menor que três, criaremos um novo projétil. No entanto, se já houver três projéteis ativos, nada acontecerá quando a barra de espaço for pressionada. Se executar o jogo agora, você deverá ser capaz de disparar projéteis somente em grupos de três.

Criando a função update_bullets()

Queremos manter nosso arquivo principal do programa, *alien_invasion.py*, o mais simples possível, portanto, agora que escrevemos e conferimos o código para gerenciamento de projéteis, podemos passá-lo para o módulo `game_functions`. Criaremos uma nova função chamada `update_bullets()` e a adicionaremos no final de *game_functions.py*:

game_functions.py

```
def update_bullets(bullets):
    """Atualiza a posição dos projéteis e se livra dos projéteis
antigos."""
    # Atualiza as posições dos projéteis
    bullets.update()
```

```
    # Livra-se dos projéteis que desapareceram
    for bullet in bullets.copy():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)
```

O código de `update_bullets()` é removido de *alien_invasion.py* e colado no novo arquivo; o único parâmetro necessário é o grupo `bullets`.

O laço `while` em *alien_invasion.py* ficou mais simples novamente:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
u       gf.check_events(ai_settings, screen, ship, bullets)
v       ship.update()
w       gf.update_bullets(bullets)
x       gf.update_screen(ai_settings, screen, ship, bullets)
```

Criamos o programa de modo que o nosso laço principal contenha apenas um mínimo de código para que possamos ler rapidamente os nomes das funções e entender o que acontece no jogo. O laço principal verifica se há entradas do jogador em u e então atualiza a posição da espaçonave em v e de qualquer projétil que tenha sido disparado em w. Em seguida usamos as posições atualizadas para desenhar uma nova tela em x.

Criando a função fire_bullet()

Vamos transferir o código de disparo de um projétil para uma função separada; assim podemos usar uma única linha de código para disparar um projétil e simplificar o bloco `elif` em `check_keydown_events()`:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    """Responde a pressionamentos de tecla."""
    --trecho omitido--
    elif event.key == pygame.K_SPACE:
        fire_bullet(ai_settings, screen, ship, bullets)

def fire_bullet(ai_settings, screen, ship, bullets):
    """Dispara um projétil se o limite ainda não foi alcançado."""
```

```
# Cria um novo projétil e o adiciona ao grupo de projéteis
if len(bullets) < ai_settings.bullets_allowed:
    new_bullet = Bullet(ai_settings, screen, ship)
    bullets.add(new_bullet)
```

A função `fire_bullet()` simplesmente contém o código usado para disparar um projétil quando a barra de espaço for pressionada; além disso, acrescentamos uma chamada a `fire_bullet()` em `check_keydown_events()` quando isso ocorrer.

Execute *alien_invasion.py* mais uma vez e certifique-se de que você ainda possa continuar atirando sem erros.

FAÇA VOCÊ MESMO

12.5 – Disparos laterais: Escreva um jogo que posicione uma espaçonave do lado esquerdo da tela e permita que o jogador a desloque para cima e para baixo. Faça a espaçonave disparar um projétil que se move para a direita da tela quando o jogador pressionar a barra de espaço. Garanta que os projéteis sejam apagados quando desaparecerem da tela.

Resumo

Neste capítulo aprendemos a criar um plano para um jogo. Conhecemos a estrutura básica de um jogo escrito com o Pygame. Vimos como definir uma cor de fundo e armazenar configurações em uma classe separada, na qual esses dados possam ser disponibilizados para todas as partes do jogo. Aprendemos a desenhar uma imagem na tela e permitir que o jogador controle o movimento dos elementos do jogo. Vimos como criar elementos que se movem por conta própria, como projéteis que se deslocam para cima na tela, e aprendemos a apagar objetos que não sejam mais necessários. Aprendemos a refatorar o código de um projeto com regularidade a fim de facilitar um desenvolvimento contínuo.

No Capítulo 13 acrescentaremos alienígenas na Invasão Alienígena. E no final desse capítulo você será capaz de eliminar os alienígenas com disparos; espero que você consiga fazer isso antes que eles alcancem a sua espaçonave.

[1] N.T.: Gameplay ou jogabilidade é um termo que se refere às experiências do jogador durante a sua interação com os sistemas de um jogo; o termo descreve aspectos como a facilidade de uso do jogo, a quantidade de vezes que ele pode ser completado ou a sua duração (baseado em: *https://pt.wikipedia.org/wiki/Jogabilidade*).

13
ALIENÍGENAS!

Neste capítulo acrescentaremos alienígenas à Invasão Alienígena. Inicialmente adicionaremos um alienígena próximo à parte superior da tela e, em seguida, vamos gerar uma frota completa deles. Faremos a frota avançar para os lados e para baixo e nos livraremos de qualquer alienígena atingido por um projétil. Por fim, limitaremos o número de espaçonaves que um jogador pode ter e encerraremos o jogo quando ele ficar sem espaçonaves.

À medida que trabalhar neste capítulo você conhecerá melhor o Pygame e verá como administrar um projeto grande. Também aprenderá a detectar colisões entre objetos do jogo, por exemplo, entre projéteis e alienígenas. Detectar colisões ajuda a definir interações entre os elementos de seus jogos: você pode confinar um personagem entre as paredes de um labirinto ou passar uma bola entre dois personagens. Também continuaremos a trabalhar com base em um plano que revisaremos ocasionalmente a fim de manter o foco de nossas sessões de escrita de código.

Antes de começar a escrever o novo código para acrescentar uma frota de alienígenas na tela, vamos analisar o projeto e atualizar o nosso plano.

Revisando o seu projeto

Ao iniciar uma nova fase do desenvolvimento em um projeto grande, é sempre uma boa ideia revisar seu plano e deixar claro o que você quer realizar com o código que está prestes a escrever. Neste capítulo vamos:

- Analisar o nosso código e determinar se precisamos refatorá-lo antes de implementar novas funcionalidades.
- Acrescentar um único alienígena no canto superior esquerdo da tela, com um espaçamento apropriado ao seu redor.
- Usar o espaçamento em torno do primeiro alienígena e o tamanho da tela como um todo para determinar quantos alienígenas cabem na tela.

Escreveremos um laço para criar alienígenas de modo a preencher a parte superior da tela.

- Fazer a frota se mover para as laterais e para baixo até que a frota toda seja atingida ou um alienígena atinja a espaçonave ou o solo. Se a frota toda for atingida, criaremos uma nova frota. Se um alienígena atingir a espaçonave ou o solo, destruiremos a espaçonave e criaremos uma nova frota.
- Limitar o número de espaçonaves que o jogador pode usar e encerrar o jogo quando ele tiver usado o seu lote de espaçonaves.

Refinaremos esse plano à medida que as funcionalidades forem implementadas, mas isso é suficiente para começar.

Revise também o código quando estiver prestes a iniciar o trabalho com um novo conjunto de funcionalidades em um projeto. Como cada nova fase do projeto geralmente deixa um projeto mais complexo, é melhor limpar um código que esteja entulhado ou que seja ineficiente.

Embora não haja muita limpeza para fazer no momento, pois fizemos refatorações à medida que avançamos, é irritante usar o mouse para fechar o jogo sempre que o executamos para testar uma nova funcionalidade. Vamos acrescentar rapidamente um atalho de teclado para terminar o jogo quando o usuário pressionar Q:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --trecho omitido--
    elif event.key == pygame.K_q:
        sys.exit()
```

Em `check_keydown_events()`, acrescentamos um novo bloco que finaliza o jogo quando Q é pressionado. É uma alteração razoavelmente segura, pois a tecla Q está longe das teclas de direção e da barra de espaço, portanto é improvável que o jogador pressione essa tecla por acidente e saia do jogo. Agora, ao testar, você poderá pressionar Q para encerrar o jogo, em vez de usar o seu mouse para fechar a janela.

Criando o primeiro alienígena

Colocar um alienígena na tela é como posicionar uma espaçonave. O comportamento de cada alienígena é controlado por uma classe chamada `Alien`, que estruturaremos de modo semelhante à classe `Ship`. Continuaremos usando imagens bitmap por questões de simplicidade. Você pode encontrar sua própria imagem para um alienígena ou usar aquela mostrada na Figura 13.1, disponível nos recursos do livro em *https://www.nostarch.com/pythoncrashcourse/*. Essa imagem tem um plano de fundo cinza, que coincide com a cor de fundo da tela. Lembre-se de salvar o arquivo com a imagem escolhida na pasta *images*.

Figura 13.1 – O alienígena que usaremos para criar a frota.

Criando a classe Alien

Agora criaremos a classe `Alien`:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """Uma classe que representa um único alienígena da frota."""

    def __init__(self, ai_settings, screen):
        """Inicializa o alienígena e define sua posição inicial."""
        super(Alien, self).__init__()
        self.screen = screen
        self.ai_settings = ai_settings

        # Carrega a imagem do alienígena e define seu atributo rect
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()
```

```
        # Inicia cada novo alienígena próximo à parte superior esquerda
  da tela
u        self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Armazena a posição exata do alienígena
        self.x = float(self.rect.x)

    def blitme(self):
        """Desenha o alienígena em sua posição atual."""
        self.screen.blit(self.image, self.rect)
```

A maior parte dessa classe é semelhante à classe `Ship`, exceto pelo posicionamento do alienígena. Posicionaremos inicialmente cada alienígena próximo ao canto superior esquerdo da tela, colocando um espaço à esquerda que seja igual à largura do alienígena e um espaço acima dele correspondente à sua altura u.

Criando uma instância do alienígena

Agora criaremos uma instância de `Alien` em *alien_invasion.py*:

alien_invasion.py

```
--trecho omitido--
from ship import Ship
from alien import Alien
import game_functions as gf

def run_game():
    --trecho omitido--
    # Cria um alienígena
    alien = Alien(ai_settings, screen)

    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(bullets)
        gf.update_screen(ai_settings, screen, ship, alien, bullets)
```

```
run_game()
```

Nesse código importamos a nova classe `Alien` e criamos uma instância dessa classe, imediatamente antes de entrar no laço principal `while`. Como ainda não mudamos a posição do alienígena, não estamos acrescentando nada novo no laço; entretanto, modificamos a chamada a `update_screen()` para lhe passar a instância `alien`.

Fazendo o alienígena aparecer na tela

Para fazer o alienígena aparecer na tela, chamamos o seu método `blitme()` em `update_screen()`:

game_functions.py

```
def update_screen(ai_settings, screen, ship, alien, bullets):
    --trecho omitido--

    # Redesenha todos os projéteis atrás da espaçonave e dos
alienígenas
    for bullet in bullets:
        bullet.draw_bullet()
    ship.blitme()
    alien.blitme()

    # Deixa a tela mais recente visível
    pygame.display.flip()
```

Desenhamos o alienígena na tela depois que a espaçonave e os projéteis foram desenhados para que os alienígenas estejam na camada superior da tela. A Figura 13.2 mostra o primeiro alienígena na tela.

Agora que o primeiro alienígena apareceu corretamente, escreveremos o código para desenhar a frota completa.

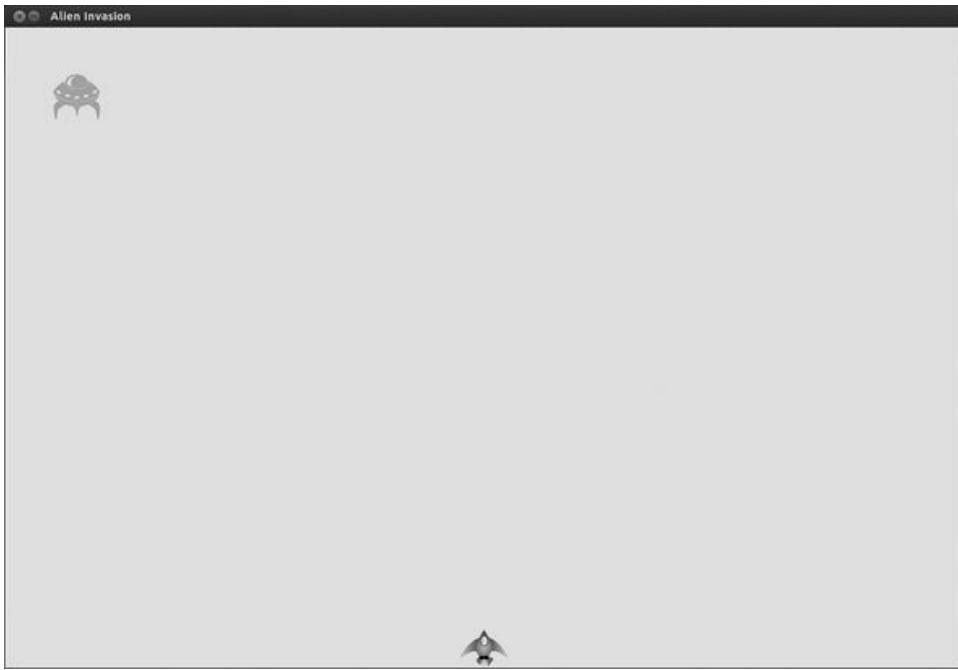



Figura 13.2 – O primeiro alienígena aparece.

Construindo a frota de alienígenas

Para desenhar uma frota, precisamos descobrir quantos alienígenas cabem na largura da tela e quantas linhas de alienígenas cabem na altura. Inicialmente determinaremos o espaçamento horizontal entre os alienígenas e criaremos uma linha; em seguida, definiremos o espaçamento vertical para criar uma frota completa.

Determinando quantos alienígenas cabem em uma linha

Para descobrir quantos alienígenas cabem em uma linha, vamos observar o espaço horizontal disponível. A largura da tela está armazenada em `ai_settings.screen_width`, mas precisamos de uma margem vazia da cada lado da tela. Faremos com que essa margem tenha a largura de um alienígena. Como temos duas margens, o espaço disponível para os alienígenas corresponde à largura da tela menos a largura de dois alienígenas:

```
available_space_x = ai_settings.screen_width - (2 * alien_width)
```

Também precisamos definir o espaçamento entre os alienígenas; faremos

com que ele tenha um alienígena de largura. O espaço necessário para exibir um alienígena é o dobro de sua largura: uma largura para o alienígena e uma para o espaço vazio à sua direita. Para descobrir quantos alienígenas cabem na largura da tela, dividimos o espaço disponível por duas vezes a largura de um alienígena:

```
number_aliens_x = available_space_x / (2 * alien_width)
```

Incluiremos esses cálculos quando criarmos a frota.

NOTA Um ótimo aspecto sobre cálculos em programação é que você não precisa ter certeza se sua fórmula está correta ao escrevê-la pela primeira vez. Você pode testá-la para ver se ela funciona. No pior caso, você terá uma tela congestionada de alienígenas ou haverá menos alienígenas que o esperado. Você pode revisar seus cálculos de acordo com o que vir na tela.

Criando linhas de alienígenas

Para gerar uma linha, crie primeiro um grupo vazio chamado `aliens` em *alien_invasion.py* para armazenar todos os nossos alienígenas e, em seguida, chame uma função em *game_functions.py* para criar uma frota:

alien_invasion.py

```
import pygame
from pygame.sprite import Group
from settings import Settings
from ship import Ship
import game_functions as gf

def run_game():
    --trecho omitido--
    # Cria uma espaçonave, um grupo de projéteis e um grupo de
alienígenas
    ship = Ship(ai_settings, screen)
    bullets = Group()
u     aliens = Group()

    # Cria a frota de alienígenas
v     gf.create_fleet(ai_settings, screen, aliens)
```

```
    # Inicia o laço principal do jogo
    while True:
        --trecho omitido--
w       gf.update_screen(ai_settings, screen, ship, aliens, bullets)

run_game()
```

Como não estamos mais criando alienígenas diretamente em *alien_invasion.py*, não será necessário importar a classe `Alien` nesse arquivo.

Crie um grupo vazio para armazenar todos os alienígenas do jogo **u**. Em seguida chame a nova função `create_fleet()` **v**, que escreveremos em breve, e passe-lhe `ai_settings`, o objeto `screen` e o grupo vazio `aliens`. Então modifique a chamada a `update_screen()` para que ela tenha acesso ao grupo de alienígenas **w**.

Também precisamos modificar `update_screen()`:

game_functions.py

```
def update_screen(ai_settings, screen, ship, aliens, bullets):
    --trecho omitido--
    ship.blitme()
    aliens.draw(screen)

    # Deixa a tela mais recente visível
    pygame.display.flip()
```

Quando `draw()` é chamado em um grupo, o Pygame desenha automaticamente cada elemento do grupo na posição definida pelo seu atributo `rect`. Nesse caso, `aliens.draw(screen)` desenhará cada alienígena do grupo na tela.

Criando a frota

Agora podemos criar a frota. A seguir, apresentamos a nova função `create_fleet()`, que colocamos no final de *game_functions.py*. Também é necessário importar a classe `Alien`, portanto lembre-se de acrescentar uma instrução `import` no início do arquivo:

game_functions.py

```
  --trecho omitido--
  from bullet import Bullet
  from alien import Alien
  --trecho omitido--

  def create_fleet(ai_settings, screen, aliens):
      """Cria uma frota completa de alienígenas."""
      # Cria um alienígena e calcula o número de alienígenas em uma linha
      # O espaçamento entre os alienígenas é igual à largura de um
  alienígena
u     alien = Alien(ai_settings, screen)
v     alien_width = alien.rect.width
w     available_space_x = ai_settings.screen_width - 2 * alien_width
x     number_aliens_x = int(available_space_x / (2 * alien_width))

      # Cria a primeira linha de alienígenas
y     for alien_number in range(number_aliens_x):
          # Cria um alienígena e o posiciona na linha
z         alien = Alien(ai_settings, screen)
          alien.x = alien_width + 2 * alien_width * alien_number
          alien.rect.x = alien.x
          aliens.add(alien)
```

Já analisamos a maior parte desse código. Precisamos conhecer a largura e a altura do alienígena para posicioná-los, portanto criamos um alienígena em u antes de fazer os cálculos. Esse alienígena não fará parte da frota, assim, não o adicione ao grupo `aliens`. Em v adquirimos a largura do alienígena a partir de seu atributo `rect` e armazenamos esse valor em `alien_width`; desse modo não precisaremos trabalhar com o atributo `rect`. Em w calculamos o espaço horizontal disponível para os alienígenas e o número de alienígenas que cabem nesse espaço.

A única mudança aqui em relação às nossas fórmulas originais está no uso de `int()` para garantir que teremos um número inteiro de alienígenas x, pois não queremos criar alienígenas parciais, e a função `range()` precisa de um inteiro. A função `int()` ignora a parte decimal de um número, fazendo o seu arredondamento para baixo. (Isso é útil porque preferimos ter um pouco de espaço extra em cada linha a ter uma linha excessivamente

congestionada.)

A seguir, defina um laço que conte de 0 até o número de alienígenas que devemos criar `y`. No corpo principal do laço, crie um novo alienígena e então defina o valor de sua coordenada x para posicioná-lo na linha `z`. Cada alienígena é inserido à direita, com um espaçamento correspondente à largura de um alienígena, a partir da margem esquerda. Em seguida, multiplicamos a largura do alienígena por dois para levar em consideração o espaço ocupado por cada alienígena, incluindo o espaço vazio à sua direita, e multiplicamos esse valor pela posição do alienígena na linha. Então adicionamos cada novo alienígena ao grupo `aliens`.

Quando executar a Invasão Alienígena, você deverá ver a primeira linha de alienígenas aparecer, como mostra a Figura 13.3.

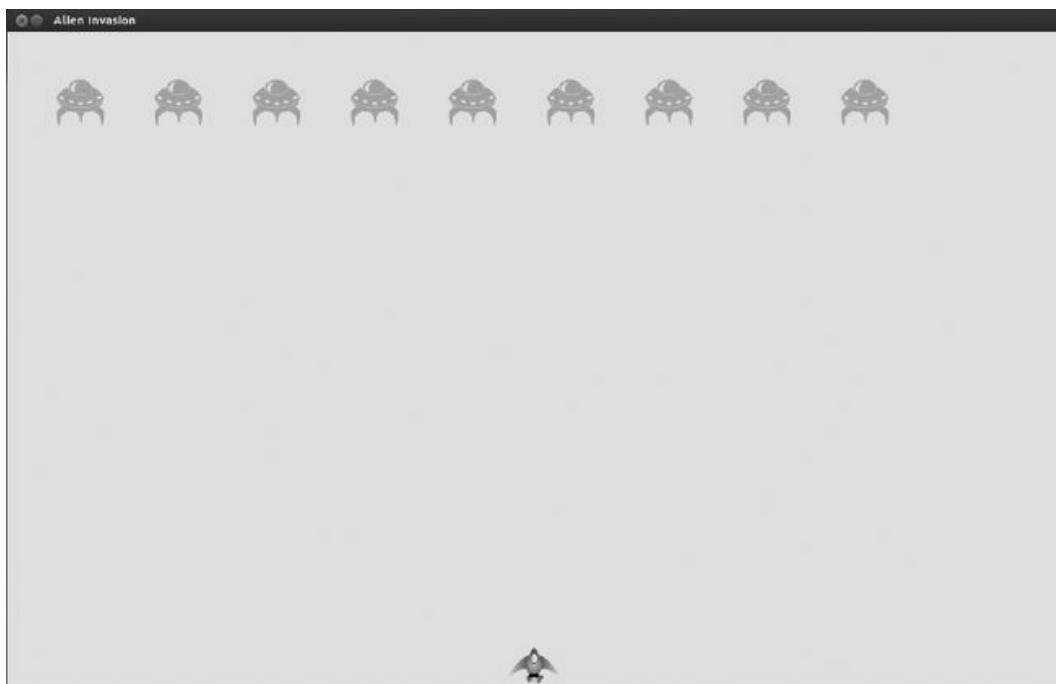



Figura 13.3 – A primeira linha de alienígenas.

A primeira linha está deslocada para a esquerda, o que, na verdade, é bom para o gameplay, pois queremos que a frota se desloque para a direita até atingir a borda da tela, depois desça um pouco e se mova para a esquerda, e assim sucessivamente. Como o jogo clássico *Space Invaders*, esse movimento é mais interessante que fazer a frota descer diretamente. Continuaremos com esse movimento até que todos os alienígenas tenham

sido atingidos ou até um alienígena atingir a espaçonave ou a parte inferior da tela.

NOTA Conforme a largura da tela que você escolher, o alinhamento da primeira linha de alienígenas poderá parecer um pouco diferente em seu sistema.

Refatorando create_fleet()

Se tivéssemos acabado de criar uma frota, provavelmente deixaríamos `create_fleet()` como está, mas ainda temos trabalho a fazer, portanto vamos limpar um pouco a função. A seguir, apresentamos `create_fleet()` com duas novas funções: `get_number_aliens_x()` e `create_alien()`:

game_functions.py

```
u def get_number_aliens_x(ai_settings, alien_width):
      """Determina o número de alienígenas que cabem em uma linha."""
      available_space_x = ai_settings.screen_width - 2 * alien_width
      number_aliens_x = int(available_space_x / (2 * alien_width))
      return number_aliens_x

  def create_alien(ai_settings, screen, aliens, alien_number):
      # Cria um alienígena e o posiciona na linha
      alien = Alien(ai_settings, screen)
v     alien_width = alien.rect.width
      alien.x = alien_width + 2 * alien_width * alien_number
      alien.rect.x = alien.x
      aliens.add(alien)

  def create_fleet(ai_settings, screen, aliens):
      """Cria uma frota completa de alienígenas."""
      # Cria um alienígena e calcula o número de alienígenas em uma linha
      alien = Alien(ai_settings, screen)
w     number_aliens_x = get_number_aliens_x(ai_settings,
alien.rect.width)

      # Cria a primeira linha de alienígenas
      for alien_number in range(number_aliens_x):
x         create_alien(ai_settings, screen, aliens, alien_number)
```

O corpo de `get_number_aliens_x()` está exatamente como era em

create_fleet() u. O corpo de create_alien() também não mudou em relação a create_fleet(), exceto que usamos o alienígena que acabou de ser criado para obter a sua largura v. Em w substituímos o código para determinar o espaçamento horizontal por uma chamada a get_number_aliens_x() e removemos a linha que referenciava alien_width, pois isso é tratado agora em create_alien(). Em x chamamos create_alien(). Essa refatoração facilitará o acréscimo de novas linha e a criação de uma frota completa.

Adicionando linhas

Para concluir a frota, determine a quantidade de linhas que cabem na tela e então repita o laço (para criar os alienígenas em uma linha) esse número de vezes. Para determinar a quantidade de linhas, calculamos o espaço vertical disponível fazendo a subtração da altura de um alienígena na parte superior, a altura da espaçonave e a altura de dois alienígenas na parte inferior da tela:

```
available_space_y = ai_settings.screen_height - 3 * alien_height -
ship_height
```

Isso resulta na criação de um espaço vazio acima da espaçonave, de modo que o jogador tenha um tempo para começar a atirar nos alienígenas no início de cada nível.

Toda linha precisa de um espaço vazio abaixo dela, que será igual à altura de um alienígena. Para calcular o número de linhas, dividimos o espaço disponível por duas vezes a altura de um alienígena. (Novamente, se esses cálculos estiverem incorretos, perceberemos de imediato e faremos ajustes até que haja um espaçamento razoável.)

```
number_rows = available_height_y / (2 * alien_height)
```

Agora que sabemos quantas linhas cabem em uma frota, podemos repetir o código para criar uma linha:

game_functions.py

```
u def get_number_rows(ai_settings, ship_height, alien_height):
```

```
    """Determina o número de linhas com alienígenas que cabem na
tela."""
v     available_space_y = (ai_settings.screen_height -
                            (3 * alien_height) - ship_height)
    number_rows = int(available_space_y / (2 * alien_height))
    return number_rows

def create_alien(ai_settings, screen, aliens, alien_number,
row_number):
    --trecho omitido--
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
w     alien.rect.y = alien.rect.height + 2 * alien.rect.height *
row_number
    aliens.add(alien)

def create_fleet(ai_settings, screen, ship, aliens):
    --trecho omitido--
    number_aliens_x = get_number_aliens_x(ai_settings,
alien.rect.width)
    number_rows = get_number_rows(ai_settings, ship.rect.height,
        alien.rect.height)

    # Cria a frota de alienígenas
x     for row_number in range(number_rows):
        for alien_number in range(number_aliens_x):
            create_alien(ai_settings, screen, aliens, alien_number,
                row_number)
```

Para calcular o número de linhas que cabem na tela, colocamos nossos cálculos de `available_space_y` e de `number_rows` na função `get_number_rows()` u, que é semelhante a `get_number_aliens_x()`. O cálculo está entre parênteses para que o resultado possa ser separado em duas linhas, o que resulta em linhas de 79 caracteres ou menos, conforme recomendado v. Usamos `int()` porque não queremos criar uma linha parcial de alienígenas.

Para criar várias linhas, usamos dois laços aninhados: um laço externo e outro interno x. O laço interno cria os alienígenas em uma linha. O laço externo conta de 0 até o número de linhas que queremos; Python usará o código para criar uma única linha e repeti-la pelo número de vezes em

number_rows.

Para aninhar os laços, escreva o novo laço for e indente o código que você deseja repetir. (A maioria dos editores de texto facilita indentar e remover a indentação de blocos de código, mas se precisar de ajuda, consulte o Apêndice B.) Agora, ao chamar create_alien(), incluímos um argumento para o número da linha para que cada linha possa ser colocada cada vez mais para baixo na tela.

A definição de create_alien() exige um parâmetro para armazenar o número da linha. Em create_alien() mudamos o valor da coordenada y de um alienígena quando ele não estiver na primeira linha W, começando com a altura de um alienígena para criar um espaço vazio na parte superior da tela. Cada linha está separada da linha anterior pela altura de dois alienígenas, portanto multiplicamos a altura do alienígena por dois e então pelo número da linha. O número da primeira linha é 0, assim o posicionamento vertical da primeira linha não muda. Todas as linhas subsequentes são colocadas cada vez mais para baixo na tela.

A definição de create_fleet() também contém um novo parâmetro para o objeto ship, o que significa que precisamos incluir o argumento ship na chamada a create_fleet() em *alien_invasion.py*:

alien_invasion.py

```
# Cria a frota de alienígenas
gf.create_fleet(ai_settings, screen, ship, aliens)
```

Ao executar o jogo agora você deverá ver uma frota de alienígenas, como mostra a Figura 13.4.

Na próxima seção faremos a frota se mover!

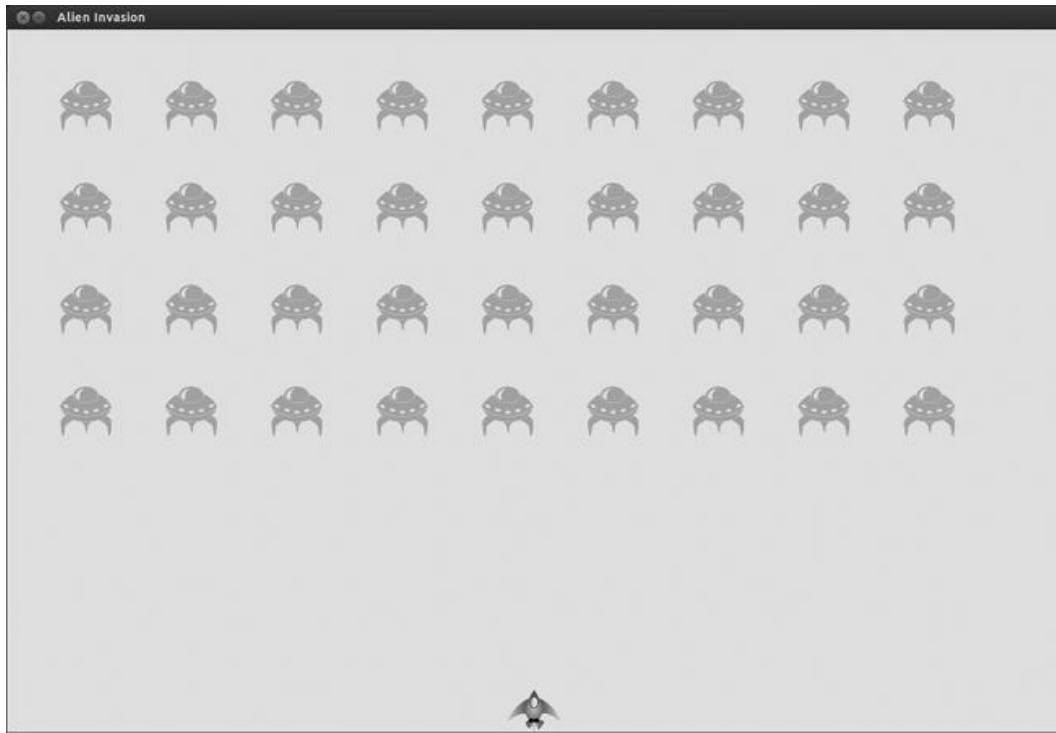



Figura 13.4 – A frota completa aparece.

FAÇA VOCÊ MESMO

13.1 – Estrelas: Encontre uma imagem de uma estrela. Faça uma grade de estrelas aparecer na tela.

13.2 – Estrelas melhoradas: Você pode criar um padrão mais realista de estrelas introduzindo uma aleatoriedade ao posicionar cada estrela. Lembre-se de que um número aleatório pode ser obtido assim:

```
from random import randint
random_number = randint(-10,10)
```

Esse código devolve um inteiro aleatório entre –10 e 10. Usando o seu código do Exercício 13.1, ajuste a posição de cada estrela de acordo com um valor aleatório.

Fazendo a frota se mover

Vamos agora fazer a nossa frota de alienígenas se mover para a direita na tela até atingir a borda e então fazê-la descer de acordo com uma distância definida e se mover para a outra direção. Continuaremos esse movimento até que todos os alienígenas tenham sido eliminados, ou um deles colida

com a espaçonave ou alcance a parte inferior da tela. Vamos começar fazendo a frota se mover para a direita.

Movendo os alienígenas para a direita

Para mover os alienígenas usaremos o método `update()` em *alien.py*, que será chamado para cada alienígena do grupo. Inicialmente adicione uma configuração para controlar a velocidade de cada alienígena:

settings.py

```
    def __init__(self):
        --trecho omitido--
        # Configurações dos alienígenas
        self.alien_speed_factor = 1
```

Então use essa configuração para implementar `update()`:

alien.py

```
    def update(self):
        """Move o alienígena para a direita."""
u       self.x += self.ai_settings.alien_speed_factor
v       self.rect.x = self.x
```

Sempre que atualizarmos a posição de um alienígena, ele será movido para a direita de acordo com o valor armazenado em `alien_speed_factor`. Controlamos a posição exata do alienígena com o atributo `self.x`, que é capaz de armazenar valores decimais u. Então usamos o valor de `self.x` para atualizar a posição do `rect` do alienígena v.

No laço principal `while`, temos chamadas para atualizar a espaçonave e os projéteis. Agora precisamos atualizar a posição de cada alienígena também:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(bullets)
        gf.update_aliens(aliens)
```

```
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Atualizamos as posições dos alienígenas depois que os projéteis foram atualizados, pois logo depois verificaremos se algum projétil atingiu um alienígena.

Por fim, adicione a nova função `update_aliens()` no final do arquivo *game_functions.py*:

game_functions.py

```
def update_aliens(aliens):
    """Atualiza as posições de todos os alienígenas da frota."""
    aliens.update()
```

Usamos o método `update()` no grupo `aliens`, o que faz o método `update()` de cada alienígena ser chamado automaticamente. Se executarmos a Invasão Alienígena agora, você deverá ver a frota se mover para a direita e desaparecer na lateral da tela.

Criando configurações para a direção da frota

Agora criaremos as configurações que farão a frota se deslocar para baixo e para a esquerda quando ela atingir a borda direita da tela. Eis o modo de implementar esse comportamento:

settings.py

```
        # Configurações dos alienígenas
        self.alien_speed_factor = 1
        self.fleet_drop_speed = 10
        # fleet_direction igual a 1 representa a direita; -1 representa
a esquerda
        self.fleet_direction = 1
```

A configuração `fleet_drop_speed` controla a velocidade com que a frota desce na tela sempre que um alienígena alcançar uma das bordas. É conveniente separar essa velocidade da velocidade horizontal dos alienígenas para que você possa ajustar as duas velocidades de modo independente.

Para implementar a configuração `fleet_direction` poderíamos ter usado

um valor textual, por exemplo, 'left' ou 'right', mas acabaríamos com instruções if-elif para testar a direção da frota. Em vez disso, como temos apenas duas direções, vamos usar os valores 1 e -1 e alternar entre eles sempre que a frota mudar de direção. (Usar números também faz sentido porque movimentar-se para a direita envolve somar um valor à coordenada x de cada alienígena, enquanto movimentar-se para a esquerda envolve fazer uma subtração no valor da coordenada x de cada alienígena.)

Verificando se um alienígena atingiu a borda

Agora precisamos de um método para verificar se um alienígena está em alguma das bordas e modificar update() para permitir que cada alienígena se desloque na direção apropriada:

alien.py

```
    def check_edges(self):
        """Devolve True se o alienígena estiver na borda da tela."""
        screen_rect = self.screen.get_rect()
u       if self.rect.right >= screen_rect.right:
            return True
v       elif self.rect.left <= 0:
            return True

    def update(self):
        """Move o alienígena para a direita ou para a esquerda."""
w       self.x += (self.ai_settings.alien_speed_factor *
                        self.ai_settings.fleet_direction)
        self.rect.x = self.x
```

Podemos chamar o novo método check_edges() em qualquer alienígena para ver se ele está na borda esquerda ou direita. O alienígena estará na borda direita se o atributo right de seu rect for maior ou igual ao atributo right do rect da tela u. Estará na borda esquerda se o valor de left for menor ou igual a 0 v.

Modificamos o método update() para permitir o movimento para a esquerda ou para a direita w multiplicando o fator de velocidade do alienígena pelo valor de fleet_direction. Se fleet_direction for 1, o valor

de alien_speed_factor será somado à posição atual do alienígena, movendo-o para a direita; se fleet_direction for -1, o valor será subtraído da posição do alienígena, movendo-o para a esquerda.

Fazendo a frota descer e mudando a direção

Quando um alienígena alcança a borda da tela, toda a frota deve descer e mudar de direção. Desse modo, precisamos fazer algumas alterações substanciais em *game_functions.py*, pois é aí que verificamos se há algum alienígena na borda esquerda ou direita da tela. Faremos isso acontecer escrevendo as funções check_fleet_edges() e change_fleet_direction(), e então modificando update_aliens():

game_functions.py

```
  def check_fleet_edges(ai_settings, aliens):
      """Responde apropriadamente se algum alienígena alcançou uma
  borda."""
u     for alien in aliens.sprites():
          if alien.check_edges():
              change_fleet_direction(ai_settings, aliens)
              break

  def change_fleet_direction(ai_settings, aliens):
      """Faz toda a frota descer e muda a sua direção."""
      for alien in aliens.sprites():
v         alien.rect.y += ai_settings.fleet_drop_speed
      ai_settings.fleet_direction *= -1

  def update_aliens(ai_settings, aliens):
      """
      Verifica se a frota está em uma das bordas
        e então atualiza as posições de todos os alienígenas da frota.
      """
w     check_fleet_edges(ai_settings, aliens)
      aliens.update()
```

Em check_fleet_edges() percorremos a frota com um laço e chamamos check_edges() em cada alienígena u. Se check_edges() devolver True, saberemos que um alienígena está em uma borda e toda a frota deverá

mudar de direção, portanto chamamos change_fleet_direction() e saímos do laço. Em change_fleet_direction() percorremos todos os alienígenas com um laço e fazemos cada um deles descer na tela usando a configuração fleet_drop_speed V; então alteramos o valor de fleet_direction multiplicando seu valor atual por -1.

Modificamos a função update_aliens() a fim de determinar se algum alienígena está em uma das bordas chamando check_fleet_edges() W. Essa função precisa de um parâmetro ai_settings, portanto incluímos um argumento para ele na chamada a update_aliens():

alien_invasion.py

```
# Inicia o laço principal do jogo
while True:
    gf.check_events(ai_settings, screen, ship, bullets)
    ship.update()
    gf.update_bullets(bullets)
    gf.update_aliens(ai_settings, aliens)
    gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Se o jogo for executado agora, a frota deverá se mover para a frente e para trás entre as bordas da tela, e descerá sempre que atingir uma das bordas. Podemos agora começar a atirar nos alienígenas e prestar atenção em qualquer um que atinja a espaçonave ou alcance a parte inferior da tela.

FAÇA VOCÊ MESMO

13.3 – Gotas de chuva: Encontre uma imagem de uma gota de chuva e crie uma grade de gotas. Faça as gotas de chuva caírem em direção à parte inferior da tela até desaparecerem.

13.4 – Chuva contínua: Modifique o código do Exercício 13.3 para que, quando uma linha de gotas d'água desaparecer na parte inferior da tela, uma nova linha apareça na parte superior e comece a cair.

Atirando nos alienígenas

Criamos nossa espaçonave e uma frota de alienígenas, mas quando os projéteis atingirem os alienígenas, eles simplesmente os atravessarão

porque não estamos verificando se há colisões. Na programação de jogos, as *colisões* ocorrem quando os elementos do jogo se sobrepõem. Para fazer os projéteis atingirem os alienígenas, usaremos o método `sprite.groupcollide()` para identificar colisões entre os membros de dois grupos.

Detectando colisões com os projéteis

Queremos saber de imediato se um projétil atingiu um alienígena para que possamos fazer o alienígena desaparecer assim que for atingido. Para isso, detectaremos colisões logo depois de atualizar a posição de um projétil.

O método `sprite.groupcollide()` compara o `rect` de cada projétil com o `rect` de cada alienígena e devolve um dicionário contendo os projéteis e os alienígenas que colidiram. Cada chave do dicionário é um projétil e o valor correspondente é o alienígena atingido. (Usaremos esse dicionário quando implementarmos um sistema de pontuação no Capítulo 14.)

Use o código a seguir para verificar se houve colisões na função `update_bullets()`:

game_functions.py

```
def update_bullets(aliens, bullets):
    """Atualiza a posição dos projéteis e se livra dos projéteis
antigos."""
    --trecho omitido--
    # Verifica se algum projétil atingiu os alienígenas
    # Em caso afirmativo, livra-se do projétil e do alienígena
    collisions = pygame.sprite.groupcollide(bullets, aliens, True,
True)
```

A nova linha que adicionamos percorre cada projétil do grupo `bullets` com um laço e, em seguida, percorre cada alienígena do grupo `aliens`. Sempre que houver uma sobreposição entre os `rects` de um projétil e de um alienígena, `groupcollide()` adicionará um par chave-valor ao dicionário devolvido. Os dois argumentos `True` dizem ao Pygame se os projéteis e os alienígenas que colidiram devem ser apagados. (Para criar um projétil altamente eficaz, com capacidade para se deslocar até a parte superior da tela, destruindo todos os alienígenas em seu caminho, você poderia definir

o primeiro argumento booleano com `False` e manter o segundo como `True`. Os alienígenas atingidos desapareceriam, mas todos os projéteis continuariam ativos até sumirem na parte superior da tela.)

O argumento `aliens` é passado na chamada a `update_bullets()`:

alien_invasion.py

```
# Inicia o laço principal do jogo
while True:
    gf.check_events(ai_settings, screen, ship, bullets)
    ship.update()
    gf.update_bullets(aliens, bullets)
    gf.update_aliens(ai_settings, aliens)
    gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Se você executar a Invasão Alienígena agora, os alienígenas que você atingir deverão desaparecer. A Figura 13.5 mostra uma frota que foi parcialmente atingida.

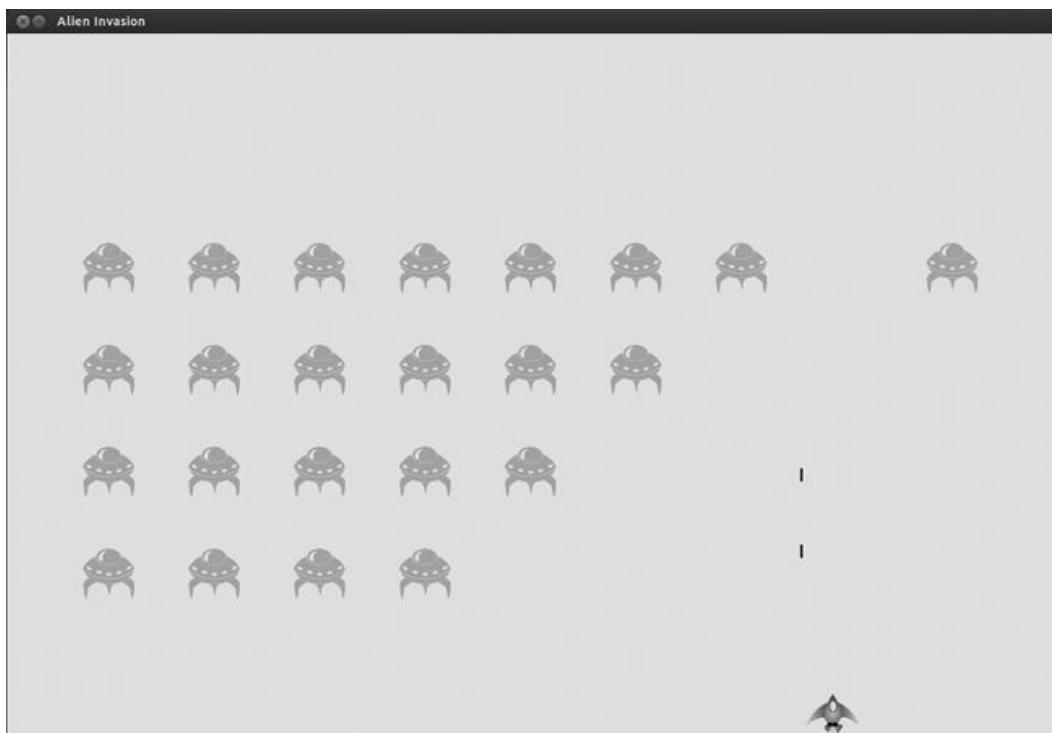
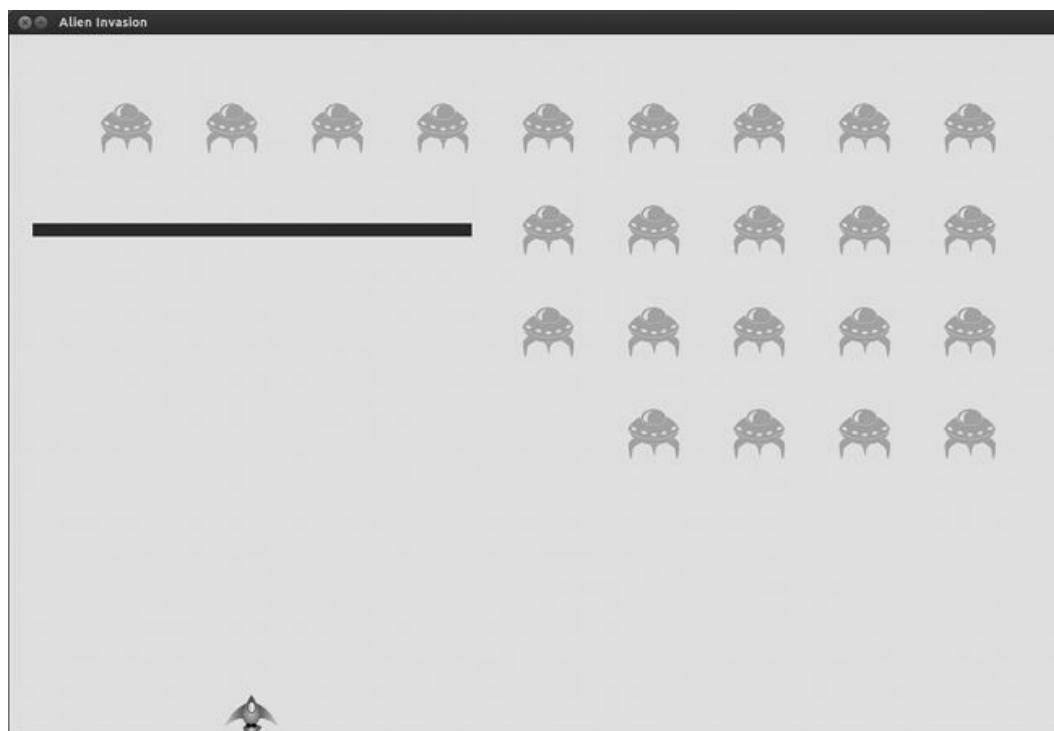



Figura 13.5 – Podemos atirar nos alienígenas!

Criando projéteis maiores para testes

Você pode testar muitas funcionalidades simplesmente executando o jogo, mas algumas delas são tediosas para testar na versão normal de um jogo. Por exemplo, é muito trabalho atingir todos os alienígenas da tela várias vezes para testar se o seu código responde a uma frota vazia de forma correta.

Para testar algumas funcionalidades em particular, você pode alterar determinadas configurações do jogo para poder se concentrar em uma área específica. Por exemplo, você pode diminuir o tamanho da tela para que haja menos alienígenas para acertar ou pode aumentar a velocidade do projétil e conceder vários projéteis de uma só vez a si mesmo.

Minha mudança preferida para testar a Invasão Alienígena é usar projéteis extremamente largos, que permaneçam ativos mesmo depois de terem atingido um alienígena (veja a Figura 13.6). Experimente configurar `bullet_width` com 300 para vez a rapidez com que você poderá eliminar a frota!

Alterações como essa ajudarão você a testar o jogo com mais eficiência e, possivelmente, lhe darão ideias para conceder bônus que aumentem a eficiência dos jogadores. (Não se esqueça de restaurar as configurações depois que terminar de testar uma funcionalidade.)

Figura 13.6 – Projéteis de alta capacidade tornam alguns aspectos do jogo mais fáceis de testar.

Repovoando a frota

Uma característica essencial da Invasão Alienígena é que os alienígenas são incansáveis: sempre que a frota for destruída, uma nova frota deverá aparecer.

Para fazer uma nova frota de alienígenas surgir depois que uma frota é destruída, verifique antes se o grupo `aliens` está vazio. Se estiver, chame `create_fleet()`. Faremos essa verificação em `update_bullets()`, pois é aí que os alienígenas individuais são destruídos:

game_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --trecho omitido--
    # Verifica se algum projétil atingiu os alienígenas
    # Em caso afirmativo, livra-se do projétil e do alienígena
    collisions = pygame.sprite.groupcollide(bullets, aliens, True,
True)

u     if len(aliens) == 0:
         # Destrói os projéteis existentes e cria uma nova frota
v         bullets.empty()
         create_fleet(ai_settings, screen, ship, aliens)
```

Em u verificamos se o grupo `aliens` está vazio. Se estiver, nos livramos de qualquer projétil existente usando o método `empty()`, que remove todos os sprites restantes de um grupo v. Também chamamos `create_fleet()`, que preenche a tela com alienígenas novamente.

A definição de `update_bullets()` agora tem os parâmetros adicionais `ai_settings`, `screen` e `ship`, portanto é necessário atualizar a chamada a essa função em *alien_invasion.py*:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
```

```
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Agora uma nova frota aparece assim que a frota atual for destruída.

Aumentando a velocidade dos projéteis

Se tentar atirar nos alienígenas no estado atual do jogo, talvez você perceba que os projéteis estão um pouco mais lentos. Isso ocorre porque o Pygame agora está executando mais tarefas a cada passagem pelo laço. Podemos aumentar a velocidade dos projéteis ajustando o valor de `bullet_speed_factor` em *settings.py*. Se aumentarmos esse valor (para 3, por exemplo), os projéteis deverão se deslocar para cima na tela a uma velocidade razoável novamente:

settings.py

```
        # Configurações dos projéteis
        self.bullet_speed_factor = 3
        self.bullet_width = 3
        --trecho omitido--
```

O melhor valor para essa configuração depende da velocidade de seu sistema, portanto determine um valor que seja adequado a você.

Refatorando update_bullets()

Vamos refatorar `update_bullets()` para que não faça tantas tarefas diferentes. Passaremos o código para lidar com colisões entre projéteis e alienígenas para outra função:

game_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --trecho omitido--
    # Livra-se dos projéteis que desapareceram
    for bullet in bullets.copy():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)
```

```
    check_bullet_alien_collisions(ai_settings, screen, ship, aliens,
bullets)

def check_bullet_alien_collisions(ai_settings, screen, ship, aliens,
bullets):
    """Responde a colisões entre projéteis e alienígenas."""
    # Remove qualquer projétil e alienígena que tenham colidido
    collisions = pygame.sprite.groupcollide(bullets, aliens, True,
True)

    if len(aliens) == 0:
        # Destrói os projéteis existentes e cria uma nova frota
        bullets.empty()
        create_fleet(ai_settings, screen, ship, aliens)
```

Criamos uma nova função, `check_bullet_alien_collisions()`, para detectar colisões entre projéteis e alienígenas, e para responder de modo apropriado caso a frota completa tenha sido destruída. Isso evita que `update_bullets()` cresça demais e simplifica futuros desenvolvimentos.

FAÇA VOCÊ MESMO

13.5 – Agarrando uma bola: Crie um jogo que posicione um personagem na parte inferior da tela; você poderá mover esse personagem para a esquerda e para a direita. Faça uma bola aparecer em uma posição aleatória na parte superior da tela e que caia a uma velocidade constante. Se seu personagem "agarrar" a bola colidindo com ela, faça a bola desaparecer. Crie uma nova bola sempre que seu personagem agarrá-la ou sempre que ela desaparecer na parte inferior da tela.

Encerrando o jogo

Qual é a graça e o desafio em um jogo se você não puder perder? Se o jogador não eliminar a frota de forma rápida o suficiente, faremos os alienígenas destruírem a espaçonave caso eles a atinjam. Ao mesmo tempo, limitaremos o número de espaçonaves que um jogador pode usar e destruiremos a espaçonave se um alienígena alcançar a parte inferior da tela. Encerraremos o jogo quando o jogador tiver usado todas as suas espaçonaves.

Detectando colisões entre um alienígena e a espaçonave

Começaremos verificando se houve colisões entre os alienígenas e a espaçonave para que possamos responder de modo apropriado quando um alienígena atingi-la. Verificaremos se houve colisões entre um alienígena e a espaçonave logo depois de atualizarmos a posição de cada alienígena:

game_functions.py

```
  def update_aliens(ai_settings, ship, aliens):
      """
      Verifica se a frota está em uma das bordas
        e então atualiza as posições de todos os alienígenas da frota.
      """
      check_fleet_edges(ai_settings, aliens)
      aliens.update()

      # Verifica se houve colisões entre alienígenas e a espaçonave
u     if pygame.sprite.spritecollideany(ship, aliens):
v         print("Ship hit!!!")
```

O método `spritecollideany()` aceita dois argumentos: um sprite e um grupo. O método verifica se algum membro do grupo colidiu com o sprite e para de percorrer o grupo assim que encontrar um membro que tenha colidido com o sprite. Nesse caso, o método percorre o grupo `aliens` e devolve o primeiro alienígena que tenha colidido com `ship`.

Se nenhuma colisão ocorreu, `spritecollideany()` devolve `None` e o bloco `if` em u não será executado. Se um alienígena que tenha colidido com a espaçonave for identificado, o método devolverá esse alienígena e o bloco `if` será executado: a mensagem *Ship hit!!!* (Espaçonave atingida!!!) será exibida v. (Quando um alienígena atinge a espaçonave, precisamos executar várias tarefas: devemos apagar todos os alienígenas e projéteis restantes, centralizar a espaçonave novamente e criar uma nova frota. Antes de escrever o código que faça tudo isso, precisamos saber se nossa abordagem para detectar colisões entre alienígenas e a espaçonave funciona corretamente. Escrever uma instrução `print` é uma maneira simples de garantir que estamos detectando colisões de modo apropriado.)

Agora `ship` deve ser passado para `update_aliens()`:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, ship, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Se executarmos a Invasão Alienígena agora, a mensagem *Ship hit!!!* deverá aparecer no terminal sempre que um alienígena colidir com a espaçonave. Ao testar essa funcionalidade, defina `alien_drop_speed` com um valor maior, por exemplo, 50 ou 100, para que os alienígenas atinjam sua espaçonave mais rapidamente.

Respondendo a colisões entre alienígenas e a espaçonave

Agora precisamos descobrir o que acontece quando um alienígena colide com a espaçonave. Em vez de destruir a instância de `ship` e criar uma nova, contaremos quantas vezes a espaçonave foi atingida armazenando dados estatísticos do jogo. (Armazenar dados estatísticos também será útil para a pontuação.)

Vamos escrever uma nova classe chamada `GameStats` para armazenar as estatísticas do jogo e salvá-la em *game_stats.py*:

game_stats.py

```
  class GameStats():
      """Armazena dados estatísticos da Invasão Alienígena."""

      def __init__(self, ai_settings):
          """Inicializa os dados estatísticos."""
          self.ai_settings = ai_settings
u         self.reset_stats()

      def reset_stats(self):
          """Inicializa os dados estatísticos que podem mudar durante o
  jogo."""
          self.ships_left = self.ai_settings.ship_limit
```

Criaremos uma instância de `GameStats` que será usada durante todo o tempo que a Invasão Alienígena executar, mas precisaremos reiniciar algumas estatísticas sempre que o jogador começar um novo jogo. Para isso, inicializaremos a maior parte dos dados estatísticos no método `reset_stats()`, e não diretamente em `__init__()`. Chamaremos esse método a partir de `__init__()` para que os dados estatísticos sejam definidos de forma apropriada quando a instância de `GameStats` é inicialmente criada u, mas também será possível chamar `reset_stats()` sempre que o jogador iniciar um novo jogo.

Nesse momento, temos apenas um dado estatístico, `ships_left`, cujo valor mudará no decorrer do jogo. O número de espaçonaves com que o jogador começa é armazenado em *settings.py* como `ship_limit`:

settings.py

```
        # Configurações da espaçonave
        self.ship_speed_factor = 1.5
        self.ship_limit = 3
```

Também devemos fazer algumas alterações em *alien_invasion.py* para criar uma instância de `GameStats`:

alien_invasion.py

```
  --trecho omitido--
  from settings import Settings
u from game_stats import GameStats
  --trecho omitido--

  def run_game():
      --trecho omitido--
      pygame.display.set_caption("Alien Invasion")

      # Cria uma instância para armazenar dados estatísticos do jogo
v     stats = GameStats(ai_settings)
      --trecho omitido--
      # Inicia o laço principal do jogo
      while True:
          --trecho omitido--
          gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
```

```
w        gf.update_aliens(ai_settings, stats, screen, ship, aliens,
bullets)
         --trecho omitido--
```

Importamos a nova classe `GameStats` u, criamos uma instância `stats` v e acrescentamos os argumentos `stats`, `screen` e `ship` na chamada a `update_aliens()` w. Usaremos esses argumentos para monitorar o número de espaçonaves que restam ao jogador e construir uma nova frota de alienígenas quando um deles atingir a espaçonave.

Quando um alienígena atinge a espaçonave, subtraímos um do número de espaçonaves restante, destruímos todos os alienígenas e projéteis, criamos uma nova frota e reposicionamos a espaçonave no meio da tela. (Também faremos uma pausa no jogo para o que o jogador possa perceber a colisão e se recompor antes que uma nova frota apareça.)

Vamos colocar a maior parte desse código na função `ship_hit()`:

game_functions.py

```
  import sys
u from time import sleep

  import pygame
  --trecho omitido--

  def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
      """Responde ao fato de a espaçonave ter sido atingida por um
  alienígena."""
      # Decrementa ships_left
v     stats.ships_left -= 1

      # Esvazia a lista de alienígenas e de projéteis
w      aliens.empty()
      bullets.empty()

      # Cria uma nova frota e centraliza a espaçonave
x     create_fleet(ai_settings, screen, ship, aliens)
      ship.center_ship()

      # Faz uma pausa
```

```
y     sleep(0.5)

z def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
      --trecho omitido--
      # Verifica se houve colisões entre alienígenas e a espaçonave
      if pygame.sprite.spritecollideany(ship, aliens):
          ship_hit(ai_settings, stats, screen, ship, aliens, bullets)
```

Inicialmente importamos a função `sleep()` do módulo `time` para fazer uma pausa no jogo u. A nova função `ship_hit()` coordena a resposta quando a espaçonave é atingida por um alienígena. Em `ship_hit()` o número de espaçonaves restante é reduzido de 1 v e, depois disso, esvaziamos os grupos `aliens` e `bullets` w.

Em seguida, criamos uma nova frota e centralizamos a espaçonave x. (Acrescentaremos o método `center_ship()` em `Ship` logo mais.) Por fim, fazemos uma pausa depois que as atualizações foram feitas em todos os elementos do jogo, mas antes de qualquer alteração ter sido desenhada na tela para que o jogador possa ver que sua espaçonave foi atingida y. A tela ficará momentaneamente congelada e o jogador verá que o alienígena atingiu a espaçonave. Quando a função `sleep()` terminar, o código continuará na função `update_screen()`, que desenhará a nova frota na tela.

Também atualizamos a definição de `update_aliens()` de modo a incluir os parâmetros `stats`, `screen` e `bullets` z; assim, esses valores poderão ser passados para a chamada a `ship_hit()`.

Eis o novo método `center_ship()`; acrescente-o no final de *ship.py*:

ship.py

```
    def center_ship(self):
        """Centraliza a espaçonave na tela."""
        self.center = self.screen_rect.centerx
```

Para centralizar a espaçonave, definimos o valor de seu atributo `center` para que coincida com o centro da tela; esse valor é obtido por meio do atributo `screen_rect`.

NOTA Observe que jamais criamos mais de uma espaçonave; criamos uma única instância da espaçonave para o jogo todo e a centralizamos novamente sempre

que ela for atingida. O dado estatístico `ships_left` nos informará quando o jogador tiver usado todas as espaçonaves.

Execute o jogo, atire em alguns alienígenas e deixe um deles atingir a espaçonave. Deverá haver uma pausa no jogo e uma nova frota aparecerá com a espaçonave centralizada na parte inferior da tela novamente.

Alienígenas que alcançam a parte inferior da tela

Se um alienígena atingir a parte inferior da tela, responderemos do mesmo modo que fizemos quando um alienígena atinge a espaçonave. Acrescente uma nova função para fazer essa verificação e chame-a a partir de `update_aliens()`:

game_functions.py

```
  def check_aliens_bottom(ai_settings, stats, screen, ship, aliens,
  bullets):
      """Verifica se algum alienígena alcançou a parte inferior da
  tela."""
      screen_rect = screen.get_rect()
      for alien in aliens.sprites():
u         if alien.rect.bottom >= screen_rect.bottom:
              # Trata esse caso do mesmo modo que é feito quando a
  espaçonave é atingida
              ship_hit(ai_settings, stats, screen, ship, aliens, bullets)
              break

  def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
      --trecho omitido--
      # Verifica se há algum alienígena que atingiu a parte inferior da
  tela
v     check_aliens_bottom(ai_settings, stats, screen, ship, aliens,
bullets)
```

A função `check_aliens_bottom()` verifica se algum alienígena alcançou a parte inferior da tela. Um alienígena alcança a parte inferior da tela quando o valor de seu `rect.bottom` for maior ou igual ao atributo `rect.bottom` da tela `u`. Se um alienígena alcançar a parte inferior da tela, chamamos `ship_hit()`. Se apenas um alienígena atingir a parte inferior, não há necessidade de verificar o restante, portanto saímos do laço depois de chamar `ship_hit()`.

Chamamos check_aliens_bottom() depois de atualizar as posições de todos os alienígenas e de verificar se houve colisões entre alienígenas e a espaçonave v. Agora uma nova frota aparecerá sempre que a espaçonave for atingida por um alienígena ou um alienígena alcançar a parte inferior da tela.

Fim de jogo!

A Invasão Alienígena parece estar mais completa agora, mas o jogo jamais termina. O valor de ships_left simplesmente assume valores negativos cada vez menores. Vamos adicionar uma flag game_active como um atributo de GameStats para encerrar o jogo quando o jogador ficar sem espaçonaves:

game_stats.py

```
    def __init__(self, settings):
        --trecho omitido--
        # Inicia a Invasão Alienígena em um estado ativo
        self.game_active = True
```

Agora acrescentamos um código em ship_hit() para definir game_active com False se o jogador usou todas as suas espaçonaves:

game_functions.py

```
def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
    """Responde ao fato de a espaçonave ter sido atingida por um
alienígena."""
    if stats.ships_left > 0:
        # Decrementa ships_left
        stats.ships_left -= 1
        --trecho omitido--
        # Faz uma pausa
        sleep(0.5)

    else:
        stats.game_active = False
```

A maior parte de ship_hit() permaneceu inalterada. Transferimos todo o código existente para um bloco if, que testa se o jogador tem pelo menos

uma espaçonave restante. Em caso afirmativo, criamos uma nova frota, fazemos uma pausa e prosseguimos. Se o jogador não tiver nenhuma espaçonave restante, definimos `game_active` com `False`.

Identificando quando determinadas partes do jogo devem executar

Em *alien_invasion.py*, precisamos identificar as partes do jogo que sempre devem executar e as partes que devem executar somente quando o jogo estiver ativo:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)

        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, ship, aliens,
  bullets)
            gf.update_aliens(ai_settings, stats, screen, ship, aliens,
  bullets)

        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

No laço principal, sempre devemos chamar `check_events()`, mesmo se o jogo estiver inativo. Por exemplo, ainda precisamos saber se o usuário pressionou Q para sair do jogo ou se clicou no botão para fechar a janela. Também continuamos atualizando a tela para que possamos fazer alterações nela enquanto esperamos para ver se o jogador optou por iniciar um novo jogo. O restante das chamadas de função só deve ocorrer quando o jogo estiver ativo, pois, se estiver inativo, não será necessário atualizar as posições dos elementos do jogo.

Agora, quando você jogar Invasão Alienígena, o jogo deverá ficar congelado quando todas as suas espaçonaves forem usadas.

FAÇA VOCÊ MESMO

13.6 – Fim de jogo: Usando o código do Exercício 13.5 (página 370), mantenha o controle do número de vezes que o jogador erra a bola. Quando ele errar a bola três vezes, encerre o jogo.

Resumo

Neste capítulo aprendemos a adicionar um grande número de elementos idênticos em um jogo criando uma frota de alienígenas. Vimos como usar laços aninhados a fim de criar uma grade de elementos, e fizemos um grande conjunto de elementos do jogo se mover chamando o método `update()` de cada elemento. Aprendemos a controlar a direção dos objetos na tela e a responder a eventos, por exemplo, quando a frota alcança a margem da tela. Também vimos como detectar e responder a colisões quando os projéteis atingem os alienígenas e esses atingem a espaçonave. Por fim, aprendemos a monitorar os dados estatísticos de um jogo e a usar uma flag `game_active` para determinar se o jogo acabou.

No último capítulo deste projeto acrescentaremos um botão Play (Jogar) para que o jogador possa decidir quando quer iniciar o seu primeiro jogo e se quer jogar novamente quando o jogo terminar. Deixaremos o jogo mais rápido sempre que o jogador atingir uma frota completa e acrescentaremos um sistema de pontuação. Como resultado, teremos um jogo totalmente funcional!

14
PONTUAÇÃO

Neste capítulo terminaremos o jogo Invasão Alienígena. Adicionaremos um botão Play para iniciar o jogo por demanda ou reiniciá-lo depois que terminar. Também mudaremos o jogo para que fique mais rápido quando o jogador passar para o próximo nível e implementaremos um sistema de pontuação. No final do capítulo você saberá o suficiente para começar a escrever jogos que aumentem o nível de dificuldade à medida que o jogador progredir e que mostrem as pontuações.

Adicionando o botão Play

Nesta seção adicionaremos um botão Play (Jogar) que aparece antes de um jogo começar e reaparece quando ele termina para que seja possível jogar novamente.

No momento, o jogo começa assim que *alien_invasion.py* é executado. Vamos iniciar o jogo em um estado inativo e então pedir que o jogador clique em um botão Play para começar. Para isso, digite o seguinte em *game_stats.py*:

game_stats.py

```
def __init__(self, ai_settings):
    """Inicializa os dados estatísticos."""
    self.ai_settings = ai_settings
    self.reset_stats()

    # Inicia o jogo em um estado inativo
    self.game_active = False

def reset_stats(self):
    --trecho omitido--
```

Agora o jogo deve começar em um estado inativo, sem uma maneira de o jogador iniciá-lo até criarmos um botão Play.

Criando uma classe Button

Como o Pygame não tem um método embutido para criar botões, escreveremos uma classe `Button` para criar um retângulo preenchido e que tenha um rótulo. Você poderá usar esse código para criar qualquer botão em um jogo. Eis a primeira parte da classe `Button`; salve-a em *button.py*:

button.py

```
  import pygame.font

  class Button():

u     def __init__(self, ai_settings, screen, msg):
          """Inicializa os atributos do botão."""
          self.screen = screen
          self.screen_rect = screen.get_rect()

          # Define as dimensões e as propriedades do botão
v         self.width, self.height = 200, 50
          self.button_color = (0, 255, 0)
          self.text_color = (255, 255, 255)
w         self.font = pygame.font.SysFont(None, 48)

          # Constrói o objeto rect do botão e o centraliza
x         self.rect = pygame.Rect(0, 0, self.width, self.height)
          self.rect.center = self.screen_rect.center

          # A mensagem do botão deve ser preparada apenas uma vez
y         self.prep_msg(msg)
```

Inicialmente importamos o módulo `pygame.font`, que permite ao Pygame renderizar um texto na tela. O método `__init__()` aceita os parâmetros `self`, os objetos `ai_settings` e `screen`, e `msg` que contém o texto do botão **u**. Estabelecemos as dimensões do botão em **v**, definimos `button_color` para colorir o objeto `rect` do botão com verde claro e definimos `text_color` para renderizar o texto em branco.

Em w preparamos um atributo font para renderizar o texto. O argumento None diz ao Pygame para usar a fonte default, e 48 determina o tamanho do texto. Para centralizar o botão na tela, criamos um rect para o botão x e definimos o seu atributo center para que seja igual ao da tela.

O Pygame trabalha com textos renderizando a string que você quer exibir como uma imagem. Em y chamamos prep_msg() para tratar essa renderização.

Eis o código de prep_msg():

button.py

```
    def prep_msg(self, msg):
        """Transforma msg em imagem renderizada e centraliza o texto no
 botão."""
u        self.msg_image = self.font.render(msg, True, self.text_color,
             self.button_color)
v        self.msg_image_rect = self.msg_image.get_rect()
         self.msg_image_rect.center = self.rect.center
```

O método prep_msg() precisa de um parâmetro self e do texto a ser renderizado como uma imagem (msg). A chamada a font.render() transforma o texto armazenado em msg em uma imagem, que então é guardada em msg_image u. O método font.render() também aceita um valor booleano para ativar ou desativar o antialiasing (o antialiasing deixa as bordas do texto mais suaves). Os argumentos restantes são a cor especificada para a fonte e a cor de fundo. Definimos o antialiasing com True e a cor de fundo do texto com a mesma cor do botão. (Se você não incluir uma cor de fundo, o Pygame tentará renderizar a fonte com uma cor de fundo transparente.)

Em v centralizamos a imagem do texto sobre o botão, criando um rect a partir da imagem e definindo seu atributo center para que seja igual ao do botão.

Por fim, criamos um método draw_button() que pode ser chamado para exibir o botão na tela:

button.py

```
    def draw_button(self):
```

```
        # Desenha um botão em branco e, em seguida, desenha a mensagem
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

Chamamos `screen.fill()` para desenhar a parte retangular do botão. Então chamamos `screen.blit()` para desenhar a imagem do texto na tela, passando-lhe uma imagem e o objeto `rect` associado a ela. Com isso, concluímos a classe `Button`.

Desenhando o botão na tela

Usaremos a classe `Button` para criar um botão `Play`. Como precisamos de apenas um botão Play, criaremos esse botão diretamente em *alien_invasion.py*, como vemos a seguir:

alien_invasion.py

```
  --trecho omitido--
  from game_stats import GameStats
  from button import Button
  --trecho omitido--

  def run_game():
      --trecho omitido--
      pygame.display.set_caption("Alien Invasion")

      # Cria o botão Play
u     play_button = Button(ai_settings, screen, "Play")
      --trecho omitido--

      # Inicia o laço principal do jogo
      while True:
          --trecho omitido--
v         gf.update_screen(ai_settings, screen, stats, ship, aliens,
bullets,
              play_button)

  run_game()
```

Importamos `Button` e criamos uma instância chamada `play_button` **u**; então passamos `play_button` para `update_screen()` para que o botão apareça

quando a tela for atualizada `v`.

Em seguida modifique `update_screen()` para que o botão Play apareça somente quando o jogo estiver inativo:

game_functions.py

```
def update_screen(ai_settings, screen, stats, ship, aliens, bullets,
play_button):
    """Atualiza as imagens na tela e alterna para a nova tela."""
    --trecho omitido--

    # Desenha o botão Play se o jogo estiver inativo
    if not stats.game_active:
        play_button.draw_button()

    # Deixa a tela mais recente visível
    pygame.display.flip()
```

Para deixar o botão Play visível sobre todos os demais elementos da tela, ele é desenhado depois que todos os outros elementos do jogo foram desenhados, mas antes de alternarmos para uma nova tela. Agora, quando executar a Invasão Alienígena, você deverá ver um botão Play no centro da tela, como mostra a Figura 14.1.

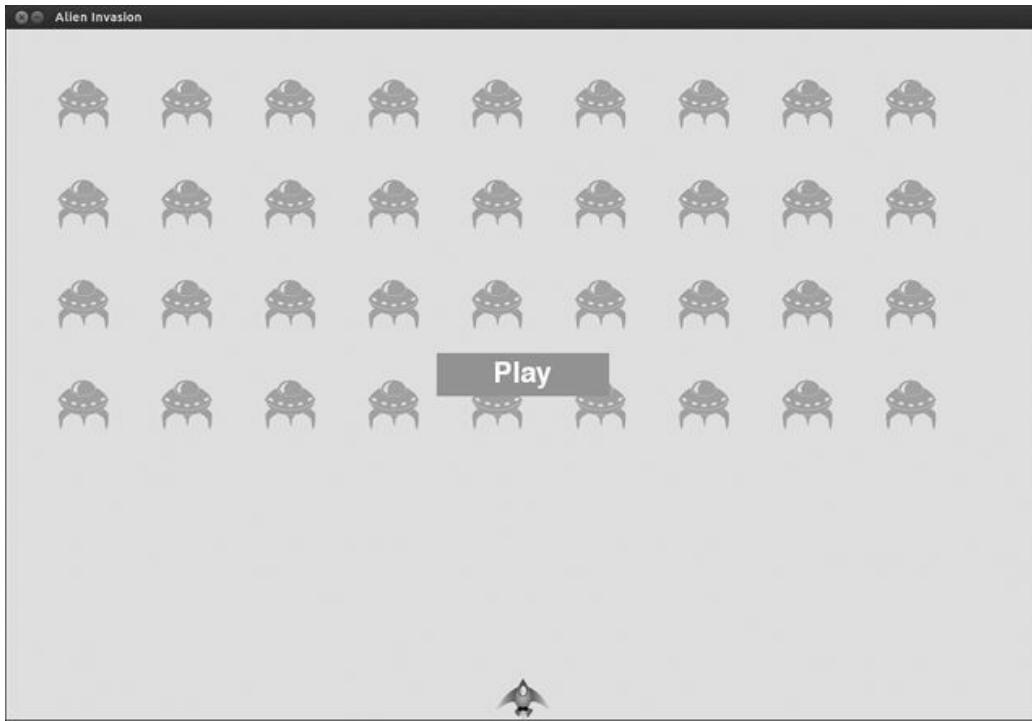



Figura 14.1 – Um botão Play aparece quando o jogo está inativo.

Iniciando o jogo

Para iniciar um novo jogo quando o jogador clicar em Play, acrescente o código a seguir em *game_functions.py* para monitorar eventos de mouse no botão:

game_functions.py

```
  def check_events(ai_settings, screen, stats, play_button, ship,
  bullets):
      """Responde a eventos de pressionamento de teclas e de mouse."""
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              --trecho omitido--
u         elif event.type == pygame.MOUSEBUTTONDOWN:
v             mouse_x, mouse_y = pygame.mouse.get_pos()
w             check_play_button(stats, play_button, mouse_x, mouse_y)

  def check_play_button(stats, play_button, mouse_x, mouse_y):
      """Inicia um novo jogo quando o jogador clicar em Play."""
x     if play_button.rect.collidepoint(mouse_x, mouse_y):
          stats.game_active = True
```

Atualizamos a definição de check_events() para que aceite os parâmetros stats e play_button. Usaremos stats para acessar a flag game_active e play_button para verificar se o botão Play foi clicado.

O Pygame detecta um evento MOUSEBUTTONDOWN quando o jogador clica em qualquer ponto da tela u, mas queremos restringir o nosso jogo de modo que ele responda a cliques do mouse somente no botão Play. Para isso, usamos pygame.mouse.get_pos(), que devolve uma tupla contendo as coordenadas x e y do cursor mouse quando o botão é clicado v. Enviamos esses valores para a função check_play_button() w, que utiliza collidepoint() para ver se o ponto em que o clique do mouse ocorreu se sobrepõe à região definida pelo rect do botão Play x. Em caso afirmativo, definimos game_active com True e o jogo começa!

A chamada a check_events() em *alien_invasion.py* deve passar dois argumentos adicionais: stats e play_button:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
bullets)
        --trecho omitido--
```

A essa altura, você deverá ser capaz de iniciar e usar um jogo completo. Quando o jogo terminar, o valor de game_active deverá se tornar False e o botão Play deverá reaparecer.

Reiniciando o jogo

O código que acabamos de escrever funciona na primeira vez que o jogador clicar em Play, mas não depois que o primeiro jogo terminar, pois as condições que fizeram o jogo ser encerrado ainda não foram reiniciadas.

Para recomeçar o jogo sempre que o jogador clicar em Play, devemos reiniciar os dados estatísticos, limpar os alienígenas e os projéteis antigos, criar uma nova frota e centralizar a espaçonave, como vemos a seguir:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship,
aliens,
        bullets, mouse_x, mouse_y):
    """Inicia um novo jogo quando o jogador clicar em Play."""
    if play_button.rect.collidepoint(mouse_x, mouse_y):
        # Reinicia os dados estatísticos do jogo
u        stats.reset_stats()
        stats.game_active = True

        # Esvazia a lista de alienígenas e de projéteis
v        aliens.empty()
        bullets.empty()

        # Cria uma nova frota e centraliza a espaçonave
w        create_fleet(ai_settings, screen, ship, aliens)
        ship.center_ship()
```

Atualizamos a definição de `check_play_button()` para que essa função tenha acesso a `ai_settings`, `stats`, `ship`, `aliens` e `bullets`. Ela precisa desses objetos para reiniciar as configurações que mudaram durante o jogo e atualizar os elementos visuais.

Em u reiniciamos os dados estatísticos do jogo, concedendo, assim, três novas espaçonaves ao jogador. Em seguida, definimos `game_active` com `True` (para que o jogo comece assim que o código dessa função acabar de executar), esvaziamos os grupos `aliens` e `bullets` v, criamos uma nova frota e centralizarmos a espaçonave w.

A definição de `check_events()` deve ser modificada, assim como a chamada a `check_play_button()`:

game_functions.py

```
def check_events(ai_settings, screen, stats, play_button, ship, aliens,
bullets):
    """Responde a eventos de pressionamento de teclas e de mouse."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --trecho omitido--
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
```

```
u             check_play_button(ai_settings, screen, stats, play_button,
ship,
                  aliens, bullets, mouse_x, mouse_y)
```

A definição de `check_events()` precisa do parâmetro `aliens`, que será passado para `check_play_button()`. Então atualizamos a chamada a `check_play_button()` para que os argumentos apropriados sejam passados **u**.

Agora atualize a chamada a `check_events()` em *alien_invasion.py* para que o argumento `aliens` seja passado:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        --trecho omitido--
```

O jogo agora será devidamente reiniciado sempre que você clicar em Play, permitindo jogá-lo quantas vezes você quiser!

Desativando o botão Play

Um problema com o nosso botão Play é que a região correspondente ao botão na tela continuará a responder a eventos de clique mesmo quando o botão não estiver mais visível. Clique na área do botão Play por acidente depois que o jogo tiver começado e ele será reiniciado!

Para corrigir isso, defina o jogo de modo que ele comece somente quando `game_active` for `False`:

game_functions.py

```
  def check_play_button(ai_settings, screen, stats, play_button, ship,
  aliens,
          bullets, mouse_x, mouse_y):
      """Inicia um novo jogo quando o jogador clicar em Play."""
u     button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
v     if button_clicked and not stats.game_active:
          # Reinicia os dados estatísticos do jogo
          --trecho omitido--
```

A flag `button_clicked` armazena um valor `True` ou `False` **u**, e o jogo será reiniciado somente se Play for clicado *e* o jogo não estiver ativo no momento **v**. Para testar esse comportamento, inicie um novo jogo e clique repetidamente no local em que o botão Play deveria estar. Se tudo funcionar conforme esperado, clicar na área do botão Play não deverá ter nenhum efeito no gameplay.

Ocultando o cursor do mouse

Queremos que o cursor do mouse esteja visível para começar a jogar, mas depois que o jogo tiver início, o cursor só atrapalhará. Para corrigir isso, vamos deixá-lo invisível depois que o jogo se tornar ativo:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship,
aliens,
        bullets, mouse_x, mouse_y):
    """Inicia um novo jogo quando o jogador clicar em Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        # Oculta o cursor do mouse
        pygame.mouse.set_visible(False)
        --trecho omitido--
```

Passar `False` para `set_visible()` diz ao Pygame para ocultar o cursor quando o mouse estiver sobre a janela do jogo.

Faremos o cursor reaparecer quando o jogo terminar para que o jogador possa clicar em Play e iniciar um novo jogo. Eis o código para fazer isso:

game_functions.py

```
def ship_hit(ai_settings, screen, stats, ship, aliens, bullets):
    """Responde ao fato de a espaçonave ter sido atingida por um
alienígena."""
    if stats.ships_left > 0:
        --trecho omitido--
    else:
        stats.game_active = False
        pygame.mouse.set_visible(True)
```

Deixamos o cursor visível novamente assim que o jogo se torna inativo, o que acontece em `ship_hit()`. A atenção a detalhes como esse deixam seu jogo parecer mais profissional e permite que o jogador se concentre em jogar, e não em entender como a interface de usuário funciona.

FAÇA VOCÊ MESMO

14.1 – Tecle P para jogar: Como a Invasão Alienígena usa entradas de teclado para controlar a espaçonave, é melhor iniciar o jogo com um pressionamento de tecla. Acrescente um código que permita ao jogador teclar P para iniciar o jogo. Transferir parte do código de `check_play_button()` para uma função `start_game()`, possível de ser chamada tanto de `check_play_button()` quanto de `check_keydown_events()`, pode ajudar.

14.2 – Treino de tiro ao alvo: Crie um retângulo na borda direita da tela que possa se mover para cima e para baixo a uma velocidade constante. Em seguida, faça uma espaçonave aparecer do lado esquerdo da tela; o jogador poderá movê-la para cima e para baixo, ao mesmo tempo que atira no retângulo em movimento. Acrescente um botão Play para iniciar o jogo e, quando o jogador errar o alvo três vezes, finalize o jogo e faça o botão Play reaparecer. Deixe o jogador reiniciar o jogo com esse botão.

Passando para o próximo nível

Em nosso jogo atual, depois que um jogador elimina toda a frota de alienígenas ele passa para um novo nível, mas a dificuldade do jogo não muda. Vamos deixar o jogo um pouco mais animado e desafiador aumentando a velocidade sempre que um jogador limpar a tela.

Modificando as configurações de velocidade

Em primeiro lugar, reorganize a classe `Settings` para agrupar as configurações do jogo em dados estáticos e dados que mudam. Também garantiremos que as configurações que mudam durante um jogo sejam reiniciadas quando um novo jogo começar. Eis o método `__init__()` de *settings.py*:

settings.py

```
def __init__(self):
```

```
        """Inicializa as configurações estáticas do jogo."""
        # Configurações da tela
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

        # Configurações da espaçonave
        self.ship_limit = 3

        # Configurações dos projéteis
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3

        # Configurações dos alienígenas
        self.fleet_drop_speed = 10

        # A taxa com que a velocidade do jogo aumenta
u       self.speedup_scale = 1.1

v       self.initialize_dynamic_settings()
```

Continuamos a inicializar as configurações que permanecem constantes no método `__init__()`. Em u acrescentamos uma configuração `speedup_scale` para controlar a taxa com que a velocidade do jogo aumenta: um valor igual a 2 dobrará a velocidade do jogo sempre que o jogador atingir um novo nível; um valor igual a 1 manterá a velocidade constante. Um valor de velocidade como 1,1 deverá fazer a velocidade aumentar o suficiente para deixar o jogo desafiador, mas não impossível. Por fim, chamamos `initialize_dynamic_settings()` para inicializar os valores dos atributos que devem mudar no curso de um jogo v.

Eis o código de `initialize_dynamic_settings()`:

settings.py

```
    def initialize_dynamic_settings(self):
        """Inicializa as configurações que mudam no decorrer do
jogo."""
        self.ship_speed_factor = 1.5
```

```
        self.bullet_speed_factor = 3
        self.alien_speed_factor = 1

        # fleet_direction igual a 1 representa a direita; -1 representa
a esquerda
        self.fleet_direction = 1
```

Esse método define os valores iniciais para as velocidades da espaçonave, dos projéteis e dos alienígenas. Aumentaremos essas velocidades à medida que o jogador fizer progressos no jogo e as reiniciaremos sempre que o jogador começar um novo jogo. Incluímos `fleet_direction` nesse método para que os alienígenas sempre se movam para a direita no início de um novo jogo. Para aumentar as velocidades da espaçonave, dos projéteis e dos alienígenas sempre que o jogador atingir um novo nível, utilize `increase_speed()`:

settings.py

```
    def increase_speed(self):
        """Aumenta as configurações de velocidade."""
        self.ship_speed_factor *= self.speedup_scale
        self.bullet_speed_factor *= self.speedup_scale
        self.alien_speed_factor *= self.speedup_scale
```

Para aumentar a velocidade desses elementos do jogo, multiplicamos cada configuração de velocidade pelo valor de `speedup_scale`.

Aumentamos o ritmo do jogo chamando `increase_speed()` em `check_bullet_alien_collisions()` quando o último alienígena da frota for atingido, mas antes de criar uma nova frota:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, ship, aliens,
bullets):
    --trecho omitido--
    if len(aliens) == 0:
        # Destrói projéteis existentes, aumenta a velocidade do jogo e
cria nova frota
        bullets.empty()
        ai_settings.increase_speed()
        create_fleet(ai_settings, screen, ship, aliens)
```

Mudar os valores das configurações de velocidade em `ship_speed_factor`, `alien_speed_factor` e `bullet_speed_factor` é suficiente para aumentar a velocidade do jogo todo!

Reiniciando a velocidade

Precisamos restaurar qualquer configuração alterada aos seus valores iniciais sempre que o jogador começar um novo jogo; caso contrário, cada novo jogo seria iniciado com as configurações de velocidade maiores, utilizadas no jogo anterior:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship,
aliens,
        bullets, mouse_x, mouse_y):
    """Inicia um novo jogo quando o jogador clicar em Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        # Reinicia as configurações do jogo
        ai_settings.initialize_dynamic_settings()

        # Oculta o cursor do mouse
        pygame.mouse.set_visible(False)
        --trecho omitido--
```

Jogar a Invasão Alienígena deverá ser mais divertido e desafiador agora. Sempre que você limpar a tela, o jogo será mais rápido e ficará um pouco mais difícil. Se o jogo se tornar difícil demais de modo muito rápido, diminua o valor de `settings.speedup_scale`; se o jogo não estiver suficientemente desafiador, aumente um pouco esse valor. Encontre um ponto ideal para aumentar a dificuldade em um período de tempo razoável. As duas primeiras telas deverão ser fáceis, a próxima deverá ser um pouco mais desafiadora, porém possível, e as telas subsequentes serão quase impossíveis.

FAÇA VOCÊ MESMO

14.3 – Tiro ao alvo desafiador: Comece com o trabalho feito no Exercício 14.2

(página 388). Faça o alvo se mover mais rápido à medida que o jogo progredir e reinicie com a velocidade original quando o jogador clicar em Play.

Pontuação

Vamos implementar um sistema de pontuação para monitorar os pontos do jogo em tempo real, assim como exibir a pontuação máxima, o nível do jogo e o número de espaçonaves restante.

A pontuação é uma estatística do jogo, portanto adicionaremos um atributo `score` em `GameStats`:

game_stats.py

```
class GameStats():
    --trecho omitido--
    def reset_stats(self):
        """Inicializa os dados estatísticos que podem mudar durante o
jogo."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

Para reiniciar a pontuação sempre que um novo jogo começar, inicializamos `score` em `reset_stats()`, e não em `__init__()`.

Exibindo a pontuação

Para exibir a pontuação na tela, inicialmente criamos uma nova classe chamada `Scoreboard`. Por enquanto, essa classe simplesmente exibirá a pontuação atual, mas nós a usaremos para informar a maior pontuação, o nível e o número de espaçonaves restante também. Eis a primeira parte da classe; salve-a em *scoreboard.py*:

scoreboard.py

```
import pygame.font

class Scoreboard():
    """Uma classe para mostrar informações sobre pontuação."""

u     def __init__(self, ai_settings, screen, stats):
```

```
        """Inicializa os atributos da pontuação."""
        self.screen = screen
        self.screen_rect = screen.get_rect()
        self.ai_settings = ai_settings
        self.stats = stats

        # Configurações de fonte para as informações de pontuação
v       self.text_color = (30, 30, 30)
w        self.font = pygame.font.SysFont(None, 48)

        # Prepara a imagem da pontuação inicial
x       self.prep_score()
```

Como `Scoreboard` escreve um texto na tela, começamos importando o módulo `pygame.font`. Em seguida, fornecemos os parâmetros `ai_settings`, `screen` e `stats` a `__init__()` para que ele possa informar os valores que estamos monitorando u. Então definimos uma cor para o texto v e instanciamos um objeto para a fonte w.

Para transformar o texto a ser exibido em uma imagem, chamamos `prep_score()` x, definido a seguir:

scoreboard.py

```
    def prep_score(self):
        """Transforma a pontuação em uma imagem renderizada."""
u       score_str = str(self.stats.score)
v       self.score_image = self.font.render(score_str, True,
self.text_color,
            self.ai_settings.bg_color)

        # Exibe a pontuação na parte superior direita da tela
w        self.score_rect = self.score_image.get_rect()
x       self.score_rect.right = self.screen_rect.right - 20
y       self.score_rect.top = 20
```

Em `prep_score()`, inicialmente transformamos o valor numérico `stats.score` em uma string u e então passamos essa string para `render()`, que cria a imagem v. Para exibir a pontuação de modo claro na tela, passamos a cor de fundo da tela para `render()`, assim como uma cor para o texto.

Posicionaremos a pontuação no canto superior direito da tela e ela será expandida para a esquerda à medida que a pontuação aumentar e a largura do número crescer. Para garantir que a pontuação esteja sempre alinhada com o lado direito da tela, criamos um `rect` chamado `score_rect` w e definimos sua borda direita a 20 pixels da borda direita da tela x. Então posicionamos a borda superior 20 pixels abaixo da parte superior da tela y.

Por fim, criamos um método `show_score()` para exibir a imagem renderizada da pontuação:

scoreboard.py

```
    def show_score(self):
        """Desenha a pontuação na tela."""
        self.screen.blit(self.score_image, self.score_rect)
```

Esse método desenha a imagem da pontuação na tela no local especificado por `score_rect`.

Criando um painel de pontuação

Para exibir a pontuação, criaremos uma instância de `Scoreboard` em *alien_invasion.py*:

alien_invasion.py

```
  --trecho omitido--
  from game_stats import GameStats
  from scoreboard import Scoreboard
  --trecho omitido--
  def run_game():
      --trecho omitido--
      # Cria instância para armazenar estatísticas do jogo e cria painel
  de pontuação
      stats = GameStats(ai_settings)
u     sb = Scoreboard(ai_settings, screen, stats)
      --trecho omitido--
      # Inicia o laço principal do jogo
      while True:
          --trecho omitido--
v         gf.update_screen(ai_settings, screen, stats, sb, ship, aliens,
```

```
        bullets, play_button)

run_game()
```

Importamos a nova classe `Scoreboard` e criamos uma instância chamada `sb` depois de criar a instância `stats` **u**. Então passamos `sb` para `update_screen()` para que a pontuação possa ser desenhada na tela **v**.

Para exibir a pontuação, modifique `update_screen()` desta maneira:

game_functions.py

```
def update_screen(ai_settings, screen, stats, sb, ship, aliens,
bullets, play_button):
    --trecho omitido--
    # Desenha a informação sobre pontuação
    sb.show_score()

    # Desenha o botão Play se o jogo estiver inativo
    if not stats.game_active:
        play_button.draw_button()

    # Deixa a tela mais recente visível
    pygame.display.flip()
```

Adicionamos `sb` à lista de parâmetros que definem `update_screen()` e chamamos `show_score()` imediatamente antes de o botão Play ser desenhado.

Quando executar a Invasão Alienígena agora, você deverá ver 0 no canto superior direito da tela. (Por enquanto, só queremos garantir que a pontuação apareça no lugar certo antes de desenvolver melhor o sistema de pontuação.) A Figura 14.2 mostra a pontuação conforme ela aparece antes de o jogo começar.

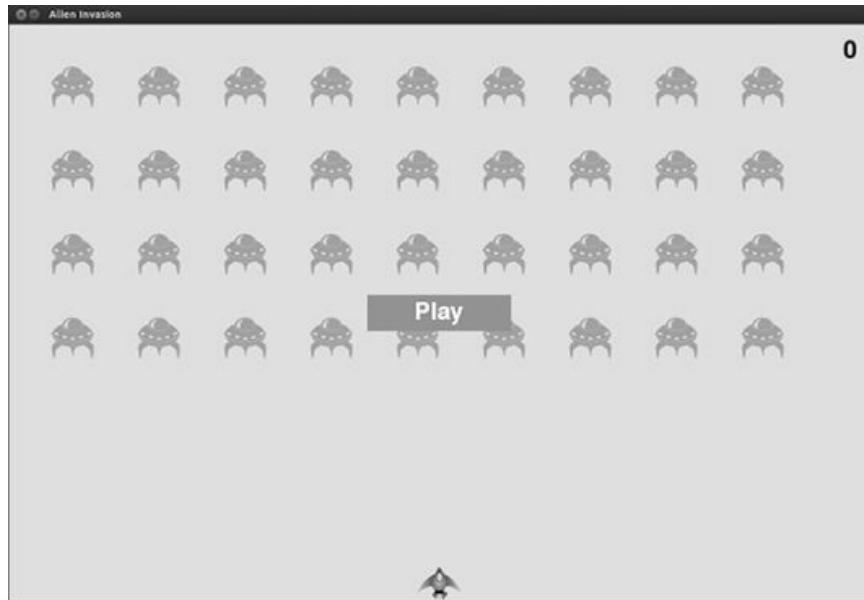



Figura 14.2 – A pontuação aparece no canto superior direito da tela.

Vamos agora atribuir pontos a cada alienígena!

Atualizando a pontuação à medida que os alienígenas são eliminados

Para termos uma pontuação em tempo real na tela, atualizamos o valor de `stats.score` sempre que um alienígena for atingido e então chamamos `prep_score()` para atualizar a imagem da pontuação. Porém, antes disso, vamos determinar quantos pontos um jogador obtém sempre que acertar um disparo em um alienígena:

settings.py

```
def initialize_dynamic_settings(self):
    --trecho omitido--

    # Pontuação
    self.alien_points = 50
```

Aumentaremos o número de pontos que cada alienígena vale à medida que o jogo progredir. Para garantir que esse valor seja reiniciado sempre que um novo jogo começar, definimos o valor em `initialize_dynamic_settings()`.

Atualize a pontuação sempre que um alienígena for atingido em `check_bullet_alien_collisions()`:

game_functions.py

```
  def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
          aliens, bullets):
      """Responde a colisões entre projéteis e alienígenas."""
      # Remove qualquer projétil e alienígena que tenham colidido
      collisions = pygame.sprite.groupcollide(bullets, aliens, True,
  True)

      if collisions:
u          stats.score += ai_settings.alien_points
          sb.prep_score()
      --trecho omitido--
```

Atualizamos a definição de `check_bullet_alien_collisions()` de modo a incluir os parâmetros `stats` e `sb`; assim a pontuação e o painel de pontuação poderão ser atualizados. Quando um projétil atinge um alienígena, o Pygame devolve um dicionário `collisions`. Verificamos se o dicionário existe e, em caso afirmativo, o valor do alienígena será somado à pontuação u. Em seguida, chamamos `prep_score()` para criar uma nova imagem da pontuação atualizada.

Precisamos modificar `update_bullets()` para garantir que os argumentos apropriados sejam passados entre as funções:

game_functions.py

```
  def update_bullets(ai_settings, screen, stats, sb, ship, aliens,
  bullets):
      """Atualiza a posição dos projéteis e se livra dos projéteis
  antigos."""
      --trecho omitido--

      check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
          aliens, bullets)
```

A definição de `update_bullets()` precisa dos parâmetros adicionais `stats` e `sb`. A chamada a `check_bullet_alien_collisions()` também deve incluir os argumentos `stats` e `sb`.

Também será necessário modificar a chamada a `update_bullets()` no laço principal `while`:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, stats, sb, ship,
aliens,
                bullets)
            --trecho omitido--
```

A chamada a `update_bullets()` precisa dos argumentos `stats` e `sb`.

Agora, quando jogar a Invasão Alienígena, você deverá ser capaz de acumular pontos!

Garantindo que todos os acertos sejam contabilizados

Como está escrito no momento, nosso código poderia deixar de contabilizar alguns alienígenas. Por exemplo, se dois projéteis colidirem com alienígenas durante a mesma passagem pelo laço ou se criarmos um projétil extragrande para atingir vários alienígenas, o jogador receberá pontos apenas por um dos alienígenas eliminados. Para corrigir isso, vamos aperfeiçoar o modo como as colisões entre alienígenas e projéteis são detectadas.

Em `check_bullet_alien_collisions()`, qualquer projétil que colidir com um alienígena se transforma em uma chave no dicionário `collisions`. O valor associado a cada projétil é uma lista de alienígenas com os quais esse projétil colidiu. Percorremos o dicionário `collisions` com um laço para garantir que concederemos pontos para cada alienígena atingido:

game_functions.py

```
 def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
         aliens, bullets):
     --trecho omitido--
     if collisions:
u        for aliens in collisions.values():
```

```
            stats.score += ai_settings.alien_points * len(aliens)
            sb.prep_score()
    --trecho omitido--
```

Se o dicionário `collisions` tiver sido definido, percorreremos todos os seus valores em um laço. Lembre-se de que cada valor é uma lista de alienígenas atingidos por um único projétil. Multiplicamos o valor de cada alienígena pelo número de alienígenas em cada lista e somamos esse valor à pontuação atual. Para testar isso, modifique a largura de um projétil para 300 pixels e confira se você recebe pontos para cada alienígena atingido com seus projéteis extragrandes; então restaure a largura do projétil de volta ao normal.

Aumentando a quantidade de pontos

Como o jogo se torna mais difícil sempre que o jogador alcança um novo nível, os alienígenas nos níveis seguintes devem valer mais pontos. Para implementar essa funcionalidade, acrescentaremos um código para aumentar a quantidade de pontos que cada alienígena vale quando a velocidade do jogo aumentar:

settings.py

```
class Settings():
    """Uma classe para armazenar todas as configurações da Invasão
Alienígena."""

    def __init__(self):
        --trecho omitido--
        # A taxa com que a velocidade do jogo aumenta
        self.speedup_scale = 1.1
        # A taxa com que os pontos para cada alienígena aumentam
u       self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def increase_speed(self):
        """Aumenta as configurações de velocidade e os pontos para cada
alienígena."""
        self.ship_speed_factor *= self.speedup_scale
```

```
        self.bullet_speed_factor *= self.speedup_scale
        self.alien_speed_factor *= self.speedup_scale

v       self.alien_points = int(self.alien_points * self.score_scale)
```

Definimos uma taxa para aumentar a quantidade de pontos, que chamamos de `score_scale` **u**. Um pequeno aumento na velocidade (1,1) deixa o jogo rapidamente mais desafiador, mas para que haja uma diferença perceptível na pontuação, você deve modificar a quantidade de pontos para cada alienígena de acordo com um fator maior (1,5). Agora, quando aumentarmos a velocidade do jogo, também aumentaremos os pontos concedidos a cada acerto **v**. Usamos a função `int()` para aumentar a quantidade de pontos com números inteiros.

Para ver quantos pontos valem cada alienígena, acrescente uma instrução `print` ao método `increase_speed()` em `Settings`:

settings.py

```
    def increase_speed(self):
        --trecho omitido--
        self.alien_points = int(self.alien_points * self.score_scale)
        print(self.alien_points)
```

Você deverá ver a nova quantidade de pontos no terminal sempre que alcançar um novo nível.

> **NOTA** Lembre-se de remover a instrução `print` depois de verificar que a quantidade de pontos está aumentando; do contrário ela poderá afetar o desempenho de seu jogo e distrair o jogador.

Arredondando a pontuação

A maioria dos jogos de tiros em estilo arcade informa as pontuações como múltiplos de dez, portanto vamos seguir essa diretriz com a nossa pontuação. Vamos também formatar a pontuação para que inclua vírgulas como separadores em números grandes. Faremos essa alteração em `Scoreboard`:

scoreboard.py

```
    def prep_score(self):
```

```
        """Transforma a pontuação em uma imagem renderizada."""
u       rounded_score = int(round(self.stats.score, -1))
v       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True,
  self.text_color,
            self.ai_settings.bg_color)
        --trecho omitido--
```

A função `round()` normalmente arredonda um número decimal com uma quantidade definida de casas decimais especificada como o segundo argumento. No entanto, se um número negativo for passado como segundo argumento, `round()` arredondará o valor para o múltiplo mais próximo de 10, 100, 1.000, e assim por diante. O código em **u** diz a Python para arredondar o valor de `stats.score` para o múltiplo mais próximo de 10 e armazená-lo em `rounded_score`.

NOTA Em Python 2.7, `round()` sempre devolve um valor decimal, portanto usamos `int()` para garantir que a pontuação seja informada como um inteiro. Se você usa Python 3, poderá remover a chamada a `int()`.

Em **v** uma diretiva para formatação de string diz a Python para inserir vírgulas nos números ao converter um valor numérico em uma string – por exemplo, para apresentar `1,000,000` em vez de `1000000`. Ao executar o jogo agora, você deverá ver uma pontuação arredondada, formatada de modo elegante, mesmo quando acumular muitos pontos, como mostra a Figura 14.3.

Figura 14.3 – Pontuação arredondada, com vírgulas como separador.

Pontuações máximas

Todo jogador quer ultrapassar a pontuação máxima de um jogo, portanto vamos monitorar e informar a pontuação máxima para oferecer um objetivo a ser visado pelos jogadores. Armazenaremos a pontuação máxima em `GameStats`:

game_stats.py

```
def __init__(self, ai_settings):
    --trecho omitido--
    # A pontuação máxima jamais deverá ser reiniciada
    self.high_score = 0
```

Como a pontuação máxima jamais deve ser reiniciada, inicializamos `high_score` em `__init__()`, e não em `reset_stats()`.

Agora modificaremos `Scoreboard` para que a pontuação máxima seja exibida. Vamos começar pelo método `__init__()`:

scoreboard.py

```
def __init__(self, ai_settings, screen, stats):
    --trecho omitido--
```

```
        # Prepara as imagens das pontuações iniciais
        self.prep_score()
u       self.prep_high_score()
```

A pontuação máxima será exibida separadamente da pontuação, portanto precisamos de um novo método, `prep_high_score()`, para preparar a imagem da pontuação máxima u.

Eis o método `prep_high_score()`:

scoreboard.py

```
    def prep_high_score(self):
        """Transforma a pontuação máxima em uma imagem renderizada."""
u       high_score = int(round(self.stats.high_score, -1))
v       high_score_str = "{:,}".format(high_score)
w       self.high_score_image = self.font.render(high_score_str, True,
            self.text_color, self.ai_settings.bg_color)

        # Centraliza a pontuação máxima na parte superior da tela
        self.high_score_rect = self.high_score_image.get_rect()
x       self.high_score_rect.centerx = self.screen_rect.centerx
y       self.high_score_rect.top = self.score_rect.top
```

Arredondamos a pontuação máxima para o múltiplo de 10 mais próximo u e a formatamos com vírgulas v. Então geramos uma imagem com a pontuação máxima w, centralizamos seu `rect` horizontalmente x e definimos seu atributo `top` para que seja igual à parte superior da imagem da pontuação y.

O método `show_score()` agora desenha a pontuação atual na parte superior à direita e a pontuação máxima no parte superior central da tela:

scoreboard.py

```
    def show_score(self):
        """Desenha a pontuação na tela."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
```

Para verificar se há uma pontuação máxima, escreveremos uma nova função, `check_high_score()`, em *game_functions.py*:

game_functions.py

```
  def check_high_score(stats, sb):
      """Verifica se há uma nova pontuação máxima."""
u     if stats.score > stats.high_score:
          stats.high_score = stats.score
          sb.prep_high_score()
```

A função `check_high_score()` aceita dois parâmetros: `stats` e `sb`. Ela usa `stats` para verificar a pontuação atual e a pontuação máxima, e precisa de `sb` para modificar a imagem da pontuação máxima quando for necessário. Em u verificamos a pontuação atual em relação à pontuação máxima. Se a pontuação atual for maior, atualizamos o valor de `high_score` e chamamos `prep_high_score()` para atualizar a imagem da pontuação máxima.

Devemos chamar `check_high_score()` sempre que um alienígena for atingido, depois de atualizar a pontuação em `check_bullet_alien_collisions()`:

game_functions.py

```
  def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
          aliens, bullets):
      --trecho omitido--
      if collisions:
          for aliens in collisions.values():
              stats.score += ai_settings.alien_points * len(aliens)
              sb.prep_score()
          check_high_score(stats, sb)
      --trecho omitido--
```

Chamamos `check_high_score()` quando o dicionário `collisions` estiver presente, e fazemos isso depois de atualizar a pontuação referente a todos os alienígenas atingidos.

Na primeira vez que você jogar a Invasão Alienígena, sua pontuação será a pontuação máxima, portanto ela será exibida como a pontuação atual e a pontuação máxima. Contudo, ao iniciar um segundo jogo, a pontuação máxima deverá aparecer no meio e a sua pontuação atual, à direita, como mostra a Figura 14.4.

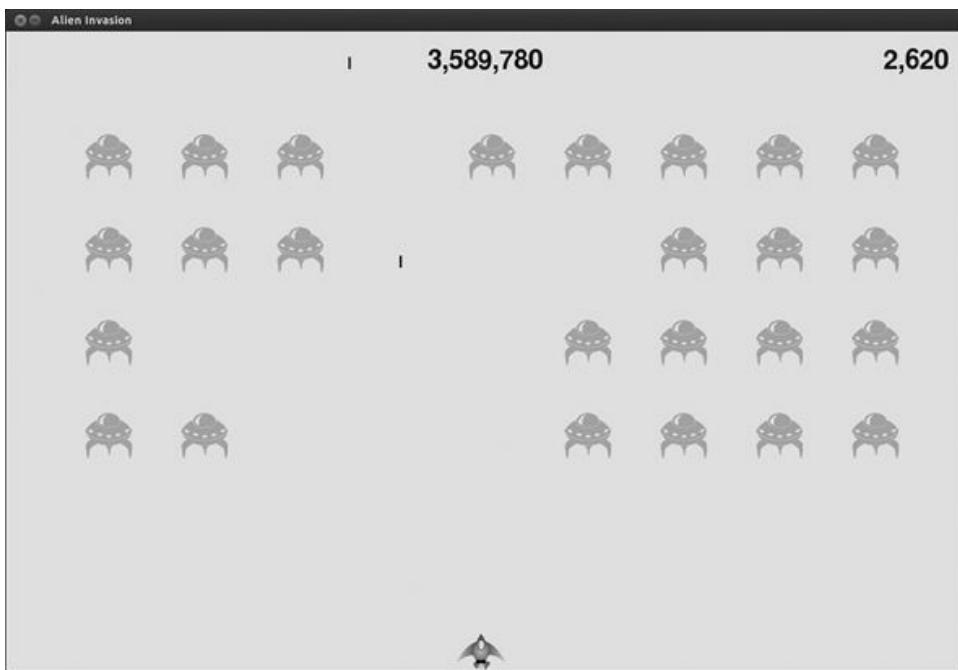



Figura 14.4 – A pontuação máxima é exibida na parte superior central da tela.

Exibindo o nível

Para exibir o nível do jogador em um jogo, precisamos antes de um atributo em `GameStats` que represente o nível atual. Para reiniciar o nível no começo de cada novo jogo, inicialize-o em `reset_stats()`:

game_stats.py

```
    def reset_stats(self):
        """Inicializa os dados estatísticos que podem mudar durante o
 jogo."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
        self.level = 1
```

Para fazer `Scoreboard` exibir o nível atual (logo abaixo da pontuação), chamamos um novo método, `prep_level()`, em `__init__()`:

scoreboard.py

```
    def __init__(self, ai_settings, screen, stats):
        --trecho omitido--

        # Prepara as imagens das pontuações iniciais
```

```
        self.prep_score()
        self.prep_high_score()
        self.prep_level()
```

Eis o método `prep_level()`:

scoreboard.py

```
    def prep_level(self):
        """Transforma o nível em uma imagem renderizada."""
u        self.level_image = self.font.render(str(self.stats.level),
True,
                self.text_color, self.ai_settings.bg_color)

        # Posiciona o nível abaixo da pontuação
        self.level_rect = self.level_image.get_rect()
v        self.level_rect.right = self.score_rect.right
w        self.level_rect.top = self.score_rect.bottom + 10
```

O método `prep_level()` cria uma imagem a partir do valor armazenado em `stats.level` u e define o atributo `right` da imagem para que seja igual ao atributo `right` da pontuação v. Então o atributo `top` é definido a 10 pixels abaixo da parte inferior da imagem da pontuação de modo a deixar um espaço entre a pontuação e o nível w.

Também devemos atualizar `show_score()`:

scoreboard.py

```
    def show_score(self):
        """Desenha as pontuações e o nível na tela."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

Esse código adiciona uma linha para desenhar a imagem do nível do jogo na tela.

Incrementaremos `stats.level` e atualizaremos a imagem do nível em `check_bullet_alien_collisions()`:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
```

```
        aliens, bullets):
    --trecho omitido--
    if len(aliens) == 0:
        # Se a frota toda for destruída, inicia um novo nível
        bullets.empty()
        ai_settings.increase_speed()

        # Aumenta o nível
u       stats.level += 1
v       sb.prep_level()

        create_fleet(ai_settings, screen, ship, aliens)
```

Se uma frota for destruída, incrementamos o valor de `stats.level` u e chamamos `prep_level()` para garantir que o novo nível seja exibido corretamente v.

Para garantir que as imagens da pontuação e do nível sejam atualizadas de forma apropriada no início de um novo jogo, dispare uma reinicialização quando o botão Play for clicado:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button, ship,
        aliens, bullets, mouse_x, mouse_y):
    """Inicia um novo jogo quando o jogador clicar em Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        --trecho omitido--

        # Reinicia os dados estatísticos do jogo
        stats.reset_stats()
        stats.game_active = True

        # Reinicia as imagens do painel de pontuação
u       sb.prep_score()
        sb.prep_high_score()
        sb.prep_level()

        # Esvazia a lista de alienígenas e de projéteis
        aliens.empty()
```

```
        bullets.empty()

        --trecho omitido--
```

A definição de `check_play_button()` precisa do objeto `sb`. Para reiniciar as imagens do painel de pontuação, chamamos `prep_score()`, `prep_high_score()` e `prep_level()` depois de reiniciar as configurações relevantes do jogo **u**.

Agora passe o parâmetro `sb` recebido por `check_events()` a `check_play_button()` para que ele tenha acesso ao objeto que representa o painel de pontuações:

game_functions.py

```
  def check_events(ai_settings, screen, stats, sb, play_button, ship,
  aliens,
          bullets):
      """Responde a eventos de pressionamento de teclas e de mouse."""
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              --trecho omitido--
          elif event.type == pygame.MOUSEBUTTONDOWN:
              mouse_x, mouse_y = pygame.mouse.get_pos()
u             check_play_button(ai_settings, screen, stats, sb,
play_button,
                  ship, aliens, bullets, mouse_x, mouse_y)
```

A definição de `check_events()` precisa receber `sb` como parâmetro para que a chamada a `check_play_button()` possa incluí-lo como argumento **u**.

Por fim, atualize a chamada a `check_events()` em *alien_invasion.py* para que `sb` também seja passado como argumento:

alien_invasion.py

```
      # Inicia o laço principal do jogo
      while True:
          gf.check_events(ai_settings, screen, stats, sb, play_button,
  ship,
              aliens, bullets)
          --trecho omitido--
```

Agora você pode ver quantos níveis foram concluídos, como mostra a Figura 14.5.

NOTA Em alguns jogos clássicos, as pontuações têm rótulos, como Score (Pontuação), High Score (Pontuação máxima) e Level (Nível). Omitimos esses rótulos porque o significado de cada número se torna evidente depois que você usar o jogo. Para incluir esses rótulos, adicione-os às strings de pontuação, imediatamente antes das chamadas a `font.render()` em Scoreboard.

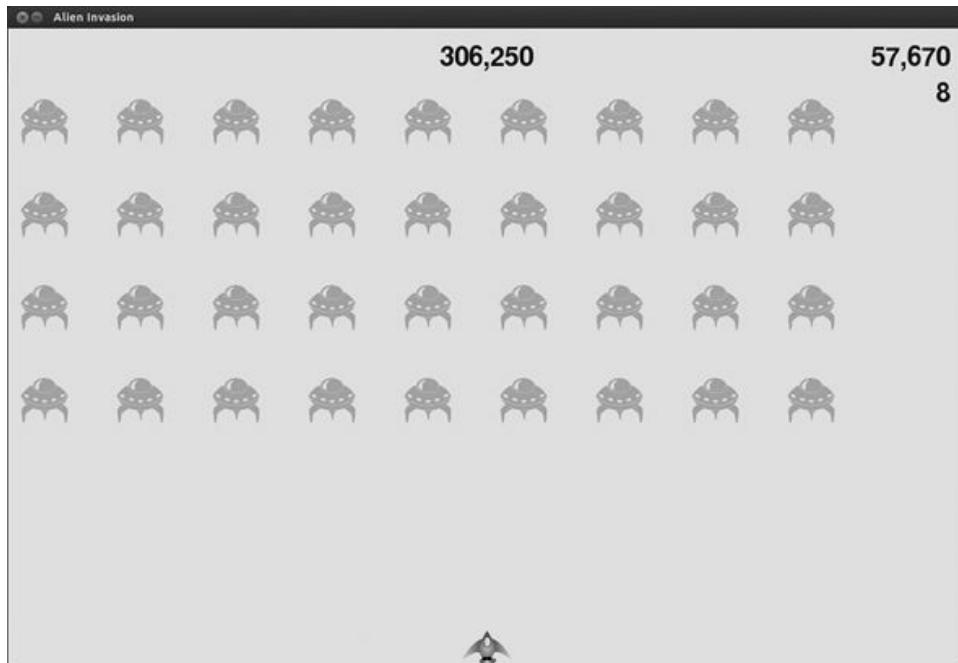



Figura 14.5 – O nível atual é informado logo abaixo da pontuação.

Exibindo o número de espaçonaves

Por fim, vamos exibir o número de espaçonaves que restam ao jogador, mas desta vez vamos usar uma imagem. Para isso, desenharemos espaçonaves no canto superior esquerdo da tela para representar quantas espaçonaves ainda restam, como em muitos jogos clássicos de arcade.

Em primeiro lugar, precisamos fazer `Ship` herdar de `Sprite` para que possamos criar um grupo de espaçonaves:

ship.py

```
import pygame
from pygame.sprite import Sprite
```

```
u class Ship(Sprite):

      def __init__(self, ai_settings, screen):
          """Inicializa a espaçonave e define sua posição inicial."""
v         super(Ship, self).__init__()
          --trecho omitido--
```

Nesse código, importamos `Sprite`, garantimos que `Ship` herde dessa classe u e chamamos `super()` no início de `__init__()` v.

Em seguida, devemos modificar `Scoreboard` a fim de criar um grupo de espaçonaves que possa ser exibido. Eis as instruções `import` e o método `__init__()`:

scoreboard.py

```
  import pygame.font
  from pygame.sprite import Group

  from ship import Ship

  class Scoreboard():
      """Uma classe para mostrar informações sobre pontuação."""

      def __init__(self, ai_settings, screen, stats):
          --trecho omitido--
          self.prep_level()
          self.prep_ships()
         --trecho omitido--
```

Como estamos criando um grupo de espaçonaves, importamos as classes `Group` e `Ship`. Chamamos `prep_ships()` após a chamada a `prep_level()`.

Eis o código de `prep_ships()`:

scoreboard.py

```
      def prep_ships(self):
          """Mostra quantas espaçonaves restam."""
u         self.ships = Group()
v         for ship_number in range(self.stats.ships_left):
              ship = Ship(self.ai_settings, self.screen)
w             ship.rect.x = 10 + ship_number * ship.rect.width
```

```
x             ship.rect.y = 10
y             self.ships.add(ship)
```

O método prep_ships() cria um grupo vazio, self.ships, para armazenar as instâncias das espaçonaves u. Para preencher esse grupo, um laço percorre todas as espaçonaves que restam ao jogador v. Nesse laço, criamos uma nova espaçonave e definimos o valor da coordenada x de cada espaçonave para que elas apareçam uma ao lado da outra, com uma margem de 10 pixels do lado esquerdo do grupo de espaçonaves w. Definimos o valor da coordenada y com 10 pixels abaixo da parte superior da tela para que as espaçonaves estejam alinhadas com a imagem da pontuação x. Por fim, adicionamos cada nova espaçonave ao grupo ships y.

Agora precisamos desenhar as espaçonaves na tela:

scoreboard.py

```
    def show_score(self):
        --trecho omitido--
        self.screen.blit(self.level_image, self.level_rect)
        # Desenha as espaçonaves
        self.ships.draw(self.screen)
```

Para exibir as espaçonaves na tela, chamamos draw() no grupo, e o Pygame desenhará cada espaçonave.

Para mostrar quantas espaçonaves um jogador tem no início, chamamos prep_ships() quando um novo jogo começar. Fazemos isso em check_play_button() no arquivo *game_functions.py*:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button,
ship,
        aliens, bullets, mouse_x, mouse_y):
    """Inicia um novo jogo quando o jogador clicar em Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        --trecho omitido--
        # Reinicia as imagens do painel de pontuação
        sb.prep_score()
        sb.prep_high_score()
```

```
        sb.prep_level()
        sb.prep_ships()
        --trecho omitido--
```

Também chamamos `prep_ships()` quando uma espaçonave é atingida para atualizar a apresentação das imagens das espaçonaves quando o jogador perder uma delas:

game_functions.py

```
u def update_aliens(ai_settings, screen, stats, sb, ship, aliens,
bullets):
    --trecho omitido--
    # Verifica se houve colisões entre alienígenas e a espaçonave
    if pygame.sprite.spritecollideany(ship, aliens):
v       ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets)

    # Verifica se há algum alienígena que atingiu a parte inferior da
  tela
w   check_aliens_bottom(ai_settings, screen, stats, sb, ship, aliens,
bullets)

x def ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets):
    """Responde ao fato de a espaçonave ter sido atingida por um
  alienígena."""
    if stats.ships_left > 0:
        # Decrementa ships_left
        stats.ships_left -= 1

        # Atualiza o painel de pontuações
y       sb.prep_ships()

        # Esvazia a lista de alienígenas e de projéteis
        --trecho omitido--
```

Inicialmente adicionamos o parâmetro `sb` à definição de `update_aliens()` u. Então passamos `sb` para `ship_hit()` v e para `check_aliens_bottom()` para que cada um deles tenha acesso ao objeto que representa o painel de pontuações w.

Em seguida atualizamos a definição de `ship_hit()` para que inclua `sb` x. Chamamos `prep_ships()` depois de decrementar o valor de `ships_left` y

para que o número correto de espaçonaves seja exibido sempre que uma delas for destruída.

Há uma chamada a `ship_hit()` em `check_aliens_bottom()`, portanto atualize essa função também:

game_functions.py

```
def check_aliens_bottom(ai_settings, screen, stats, sb, ship, aliens,
bullets):
    """Verifica se algum alienígena alcançou a parte inferior da
tela."""
    screen_rect = screen.get_rect()
    for alien in aliens.sprites():
        if alien.rect.bottom >= screen_rect.bottom:
            # Trata esse caso do mesmo modo que é feito quando uma
espaçonave é atingida
            ship_hit(ai_settings, screen, stats, sb, ship, aliens,
bullets)
            break
```

Agora `check_aliens_bottom()` aceita `sb` como parâmetro, e acrescentamos um argumento `sb` na chamada a `ship_hit()`.

Por fim, passe `sb` na chamada a `update_aliens()` em *alien_invasion.py*:

alien_invasion.py

```
    # Inicia o laço principal do jogo
    while True:
        --trecho omitido--
        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, stats, sb, ship,
aliens,
                bullets)
            gf.update_aliens(ai_settings, screen, stats, sb, ship,
aliens,
                bullets)
            --trecho omitido--
```

A Figura 14.6 mostra o sistema de pontuação completo, com as espaçonaves restantes exibidas na parte superior esquerda da tela.

Figura 14.6 – O sistema de pontuação completo da Invasão Alienígena.

FAÇA VOCÊ MESMO

14.4 – A pontuação máxima de todos os tempos: A pontuação máxima é reiniciada sempre que um jogador fecha e reinicia a Invasão Alienígena. Corrija isso gravando a pontuação máxima em um arquivo antes de chamar `sys.exit()` e lendo-a quando inicializar seu valor em `GameStats`.

14.5 – Refatoração: Procure funções e métodos que executem mais de uma tarefa e refatore-os para manter o seu código organizado e eficiente. Por exemplo, passe uma parte do código de `check_bullet_alien_collisions()`, que inicia um novo nível quando a frota de alienígenas é destruída, para uma função chamada `start_new_level()`. Além disso, transfira as quatro chamadas de método separadas no método `__init__()` em `Scoreboard` para um método chamado `prep_images()` para deixar `__init__()` mais conciso. O método `prep_images()` também poderá ajudar `check_play_button()` ou `start_game()` se você já tiver refatorado `check_play_button()`.

NOTA: *Antes de tentar refatorar o projeto, consulte o Apêndice D para aprender a restaurá-lo a um estado funcional caso você introduza bugs na refatoração.*

14.6 – Expandindo a Invasão Alienígena: Pense em um modo de expandir a Invasão Alienígena. Por exemplo, você poderia programar os alienígenas para atirar na espaçonave ou acrescentar escudos para a sua espaçonave se esconder atrás deles e que possam ser destruídos por projéteis de qualquer um dos lados.

Você também pode usar um recurso como o módulo `pygame.mixer` para acrescentar efeitos sonoros como sons de explosões e de tiros.

Resumo

Neste capítulo aprendemos a criar um botão Play para iniciar um novo jogo e a detectar eventos de movimento do mouse, além de ocultar o cursor em jogos ativos. Você pode usar o que aprendeu para criar outros botões em seus jogos, por exemplo, um botão Help (Ajuda) para exibir instruções sobre como jogar. Também aprendemos a modificar a velocidade de um jogo à medida que ele progredir, vimos como implementar um sistema progressivo de pontuação e como exibir informações de forma textual e não textual.

PROJETO 2

VISUALIZAÇÃO DE DADOS

15
GERANDO DADOS

A visualização de dados envolve a exploração de dados por meio de representações visuais. Ela está intimamente relacionada ao *data mining* (mineração de dados), que usa código para explorar padrões e conexões em um conjunto de dados. Um conjunto de dados pode ser apenas uma pequena lista de números que caiba em uma linha de código ou podem ser muitos gigabytes de dados.

Criar belas representações de dados vai muito além de gerar imagens bonitas. Quando você tem uma representação simples e visualmente atraente de um conjunto de dados, seu significado se torna evidente a quem os vê. As pessoas perceberão padrões e significados em seus conjuntos de dados, que elas nem sequer sabiam que existiam.

Felizmente não é preciso ter um supercomputador para visualizar dados complexos. Com a eficiência de Python, você poderá rapidamente explorar conjuntos de dados compostos de milhões de pontos individuais usando um notebook. Os pontos de dados não precisam ser números. Com o básico que você aprendeu na primeira parte deste livro, também será possível analisar dados não numéricos.

As pessoas usam Python para tarefas que trabalham intensamente com dados, como genética, pesquisas sobre o clima, análises políticas e econômicas e muito mais. Os cientistas de dados escreveram um conjunto impressionante de ferramentas para visualização e análise em Python, muitas das quais estão disponíveis a você também. Uma das ferramentas mais populares é o matplotlib, que é uma biblioteca matemática para construção de gráficos. Usaremos o matplotlib para a criação de gráficos simples, como gráficos lineares e de dispersão. Depois disso, criaremos um conjunto de dados mais interessante baseado no conceito de passeio aleatório (random walk) – uma visualização gerada a partir de uma série de decisões aleatórias.

Também usaremos um pacote chamado Pygal, cujo enfoque está na

criação de visualizações que funcionem bem em dispositivos digitais. Você pode usar o Pygal para enfatizar e redimensionar elementos à medida que o usuário interagir com a sua visualização, e poderá redimensionar facilmente toda a representação para que ela caiba em um minúsculo smartwatch ou em um monitor gigante. Usaremos o Pygal para explorar o que acontece se você lançar dados de várias maneiras.

Instalando o matplotlib

Em primeiro lugar, você precisará instalar o matplotlib que usaremos para o nosso conjunto inicial de visualizações. Se você ainda não usou o pip, consulte a seção "Instalando pacotes Python com o pip".

No Linux

Se você usa a versão de Python que veio com o seu sistema, poderá usar o gerenciador de pacotes que o acompanha para instalar o matplotlib utilizando apenas uma linha:

```
$ sudo apt-get install python3-matplotlib
```

Se usar Python 2.7, utilize esta linha:

```
$ sudo apt-get install python-matplotlib
```

Se você instalou uma versão mais recente de Python, será necessário instalar algumas bibliotecas das quais o matplotlib depende:

```
$ sudo apt-get install python3.5-dev python3.5-tk tk-dev
$ sudo apt-get install libfreetype6-dev g++
```

Então use o pip para instalar o matplotlib:

```
$ pip install --user matplotlib
```

No OS X

A Apple inclui o matplotlib em sua instalação-padrão de Python. Para verificar se ele está instalado em seu sistema, abra uma sessão de terminal e experimente executar `import matplotlib`. Se o matplotlib não estiver em seu

sistema ainda e você usou o Homebrew para instalar Python, instale-o assim:

```
$ pip install --user matplotlib
```

NOTA Talvez você precise usar `pip3` em vez de `pip` ao instalar os pacotes. Além disso, se esse comando não funcionar, talvez seja necessário remover a flag `--user`.

No Windows

No Windows você precisará instalar o Visual Studio antes. Acesse *https://dev.windows.com/*, clique em **Downloads** e procure o Visual Studio Community, que é um conjunto gratuito de ferramentas de desenvolvedor para Windows. Faça o download do instalador e o execute.

A seguir você precisará de um instalador para o matplotlib. Acesse *https://pypi.python.org/pypi/matplotlib/* e procure um arquivo wheel (um arquivo que termina com *.whl*) que corresponda à versão de Python que você usa. Por exemplo, se você utiliza uma versão de Python 3.5 para 32 bits, será necessário fazer o download de *matplotlib-1.4.3-cp35-none-win32.whl*.

NOTA Se você não vir um arquivo que corresponda à sua versão de Python instalada, veja o que está disponível em *http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib*. Esse site tende a disponibilizar instaladores um pouco antes do site oficial do matplotlib.

Copie o arquivo *.whl* para a sua pasta de projeto, abra uma janela de comandos e navegue até essa pasta. Então use o pip para instalar o matplotlib:

```
> cd python_work
python_work> python -m pip install --user matplotlib-1.4.3-cp35-none-win32.whl
```

Testando o matplotlib

Depois de ter instalado os pacotes necessários, teste sua instalação iniciando uma sessão de terminal com o comando **`python`** ou **`python3`** e importando o matplotlib:

```
$ python3
>>> import matplotlib
>>>
```

Se não houver nenhuma mensagem de erro, é sinal de que o matplotlib foi instalado em seu sistema e você poderá prosseguir para a próxima seção.

NOTA Se tiver problemas com a sua instalação, consulte o Apêndice C. Se tudo o mais falhar, peça ajuda. É bem provável que seu problema seja algo que um programador Python experiente poderá resolver rapidamente depois que você lhe der algumas informações.

A galeria do matplotlib

Para ver os tipos de visualizações que podem ser criadas com o matplotlib, acesse a galeria de amostras em *http://matplotlib.org/*. Ao clicar em uma visualização da galeria, você poderá ver o código usado para gerar o gráfico.

Gerando um gráfico linear simples

Vamos gerar um gráfico linear simples usando o matplotlib e então personalizá-lo a fim de criar uma visualização mais informativa de nossos dados. Usaremos a sequência 1, 4, 9, 16, 25 de números elevados ao quadrado como dados para o gráfico.

Basta fornecer os números ao matplotlib como mostramos a seguir e ele fará o resto:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
plt.plot(squares)
plt.show()
```

Inicialmente importamos `pyplot` usando o alias `plt` para que não seja necessário digitar `pyplot` repetidamente. (Você verá essa convenção com frequência em exemplos online, portanto faremos o mesmo aqui.) O

módulo `pyplot` contém várias funções que ajudam a gerar gráficos e plotagens.

Criamos uma lista para armazenar os quadrados e então a passamos para a função `plot()`, que tentará plotar os números de forma significativa. `plt.show()` abre o visualizador do matplotlib e exibe o gráfico, conforme mostrado na Figura 15.1. O visualizador permite fazer zoom e navegar pelo gráfico; se você clicar no ícone do disco, poderá salvar a imagem de qualquer gráfico que quiser.

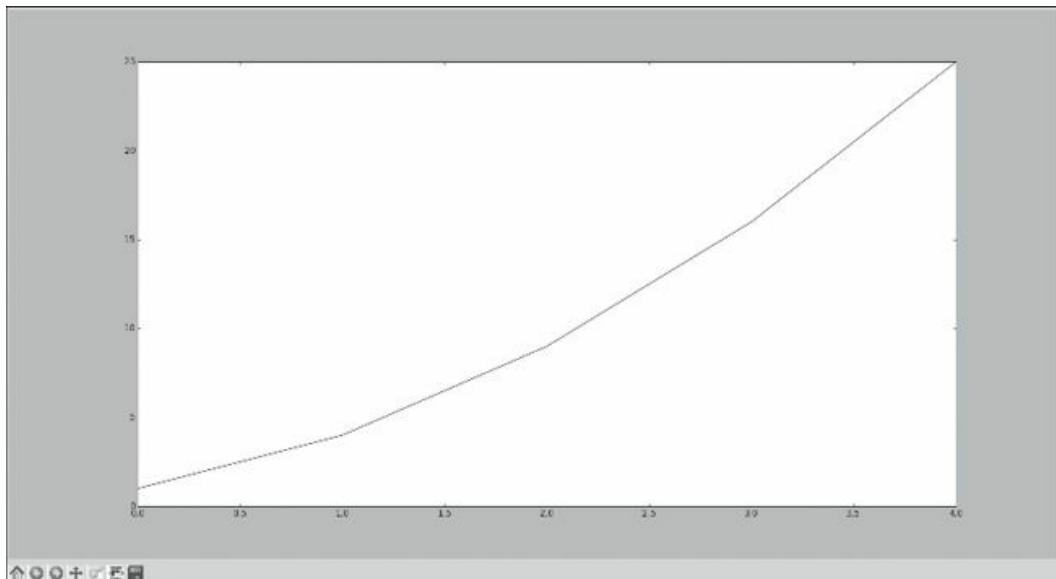



Figura 15.1 – Um dos gráficos mais simples que você pode criar no matplotlib.

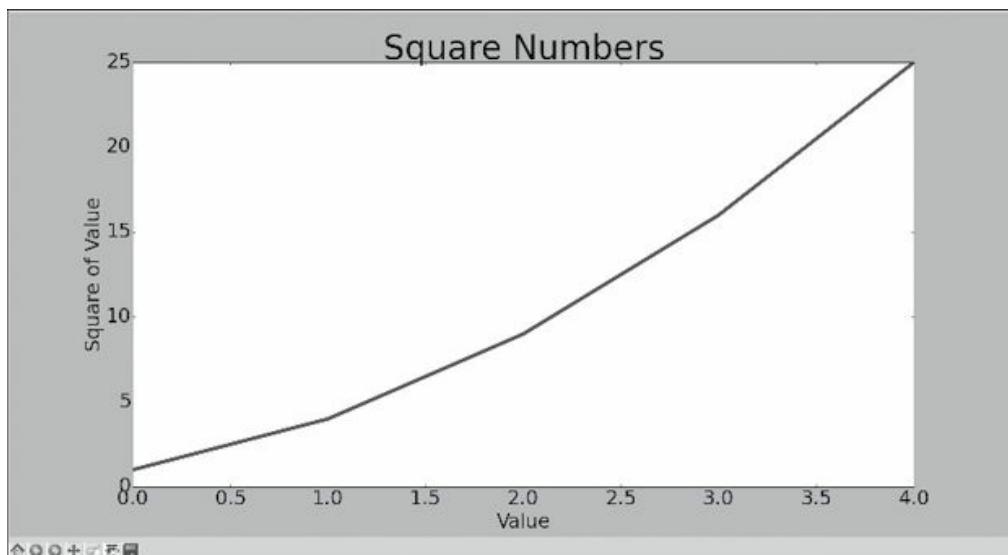
Alterando o tipo do rótulo e a espessura do gráfico

Embora o gráfico exibido na Figura 15.1 mostre que os números estão aumentando, o tipo de letra do rótulo é pequeno demais e a linha é muito fina. Felizmente o matplotlib permite ajustar todos os recursos de uma visualização.

Usaremos algumas das personalizações disponíveis para melhorar a legibilidade desse gráfico, como vemos a seguir:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
```

```
u plt.plot(squares, linewidth=5)

  # Define o título do gráfico e nomeia os eixos
v plt.title("Square Numbers", fontsize=24)
w plt.xlabel("Value", fontsize=14)
  plt.ylabel("Square of Value", fontsize=14)

  # Define o tamanho dos rótulos das marcações
x plt.tick_params(axis='both', labelsize=14)

  plt.show()
```

O parâmetro `linewidth` em u controla a espessura da linha gerada por `plot()`. A função `title()` em v define um título para o gráfico. Os parâmetros `fontsize`, que aparecem repetidamente pelo código, controlam o tamanho do texto no gráfico.

As funções `xlabel()` e `ylabel()` permitem definir um título para cada um dos eixos w, e a função `tick_params()` estiliza as marcações nos eixos x. Os argumentos mostrados aqui afetam as marcações tanto no eixo x quanto no eixo y (`axes='both'`) e definem o tamanho da fonte dos rótulos das marcações com 14 (`labelsize=14`).

Como podemos ver na Figura 15.2, o gráfico resultante é muito mais fácil de ler. O tipo de letra dos rótulos é maior e a linha do gráfico é mais espessa.

Figura 15.2 – O gráfico é muito mais fácil de ler agora.

Corrigindo o gráfico

Agora que podemos ler o gráfico mais facilmente, vemos que os dados não estão plotados corretamente. Observe no final do gráfico que o quadrado de 4.0 é mostrado como 25! Vamos corrigir isso.

Quando fornecemos uma sequência de números a `plot()`, ele supõe que o primeiro ponto de dado corresponde a um valor de coordenada x igual a 0, mas nosso primeiro ponto corresponde a um valor de x igual a 1. Podemos sobrescrever o comportamento-padrão fornecendo a `plot()` os valores tanto de entrada quanto de saída usados para calcular os quadrados:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]
plt.plot(input_values, squares, linewidth=5)

# Define o título do gráfico e nomeia os eixos
--trecho omitido--
```

Agora `plot()` colocará os dados corretamente no gráfico, pois fornecemos tanto os valores de entrada quanto os valores de saída, de modo que ele não precisou pressupor de que modo os números de saída foram gerados. O gráfico resultante, mostrado na Figura 15.3, está correto.

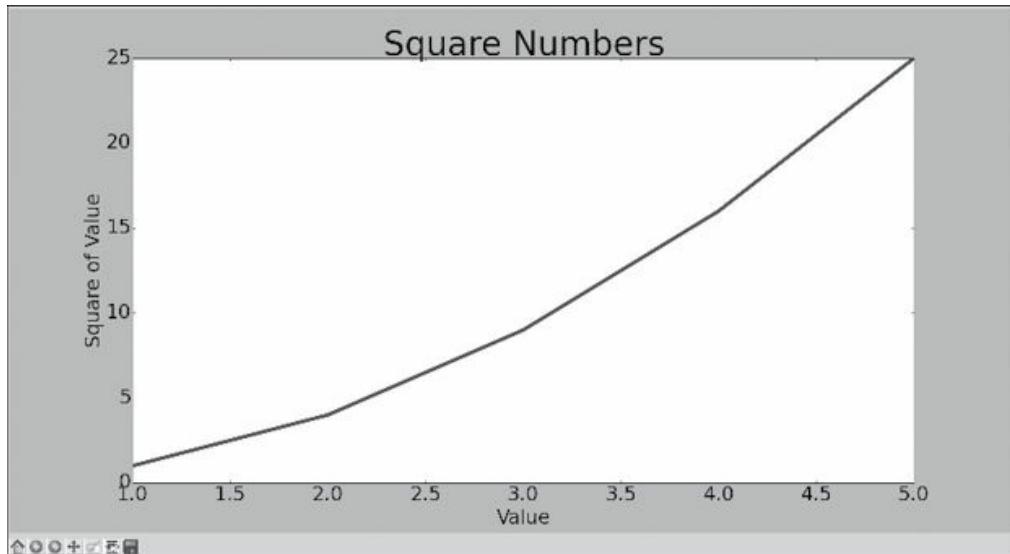



Figura 15.3 – Os dados agora foram plotados corretamente.

Podemos especificar vários argumentos ao usar `plot()` e utilizar diversas funções para personalizar seus gráficos. Continuaremos a explorar essas funções de personalização à medida que trabalharmos com conjuntos de dados mais interessantes ao longo deste capítulo.

Plotando e estilizando pontos individuais com scatter()

Às vezes é conveniente ser capaz de plotar e estilizar pontos individuais de acordo com determinadas características. Por exemplo, você pode plotar valores menores com uma cor e valores maiores com outra cor. Você também poderia plotar um conjunto grande de dados com um grupo de opções de estilização e então enfatizar pontos individuais refazendo a plotagem desses pontos com opções diferentes.

Para plotar um único ponto, utilize a função `scatter()`. Passe o único par (x, y) do ponto em que você estiver interessado para `scatter()`, e esse valor deverá ser plotado:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.scatter(2, 4)
plt.show()
```

Vamos estilizar a saída para deixá-la mais interessante. Adicionaremos

um título, nomearemos os eixos e garantiremos que todo o texto seja grande o suficiente para ser lido:

```
  import matplotlib.pyplot as plt

u plt.scatter(2, 4, s=200)

  # Define o título do gráfico e nomeia os eixos
  plt.title("Square Numbers", fontsize=24)
  plt.xlabel("Value", fontsize=14)
  plt.ylabel("Square of Value", fontsize=14)

  # Define o tamanho dos rótulos das marcações
  plt.tick_params(axis='both', which='major', labelsize=14)

  plt.show()
```

Em u chamamos `scatter()` e usamos o argumento `s` para definir o tamanho dos pontos usados para desenhar o gráfico. Ao executar *scatter_squares.py* agora, você deverá ver um único ponto no meio do gráfico, como mostra a Figura 15.4.

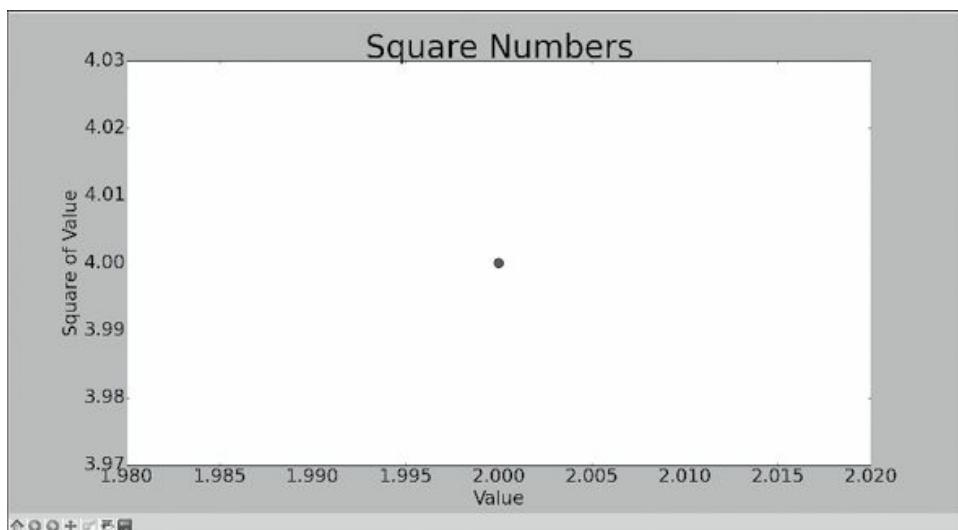



Figura 15.4 – Plotando um único ponto.

Plotando uma série de pontos com scatter()

Para plotar uma série de pontos, podemos passar listas separadas de valores para x e y para `scatter()`, assim:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.scatter(x_values, y_values, s=100)

# Define o título do gráfico e nomeia os eixos
--trecho omitido--
```

A lista `x_values` contém os números que serão elevados ao quadrado e `y_values` contém o quadrado de cada número. Quando essas listas são passadas para `scatter()`, o matplotlib lê um valor de cada lista à medida que plotar cada ponto. Os pontos a serem plotados são (1, 1), (2, 4), (3, 9), (4, 16) e (5, 25); o resultado pode ser visto na Figura 15.5.

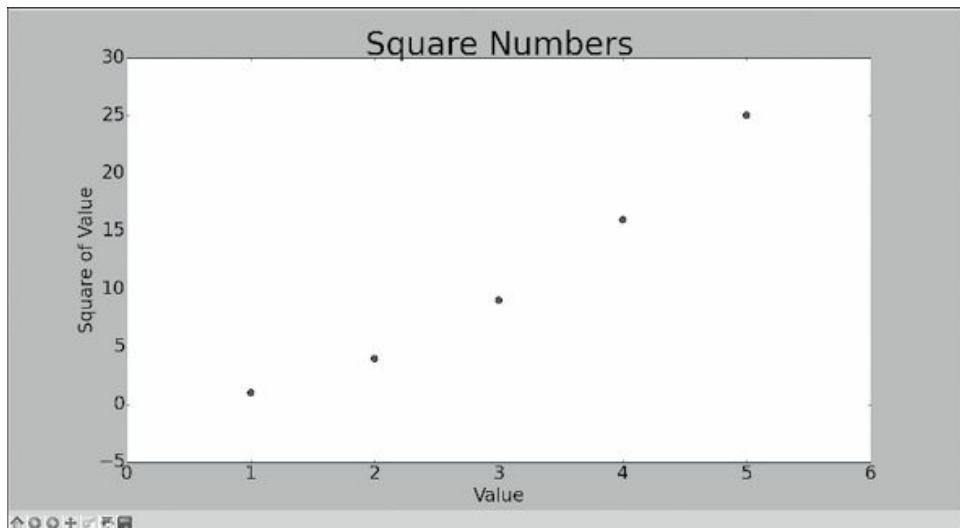



Figura 15.5 – Um gráfico de dispersão com vários pontos.

Calculando dados automaticamente

Escrever nossas listas manualmente pode ser ineficiente, em especial quando temos muitos pontos. Em vez de passar nossos pontos por meio de uma lista, vamos usar um laço em Python para que os cálculos sejam feitos para nós. Eis a aparência do código com 1.000 pontos:

scatter_squares.py

```
  import matplotlib.pyplot as plt

u x_values = list(range(1, 1001))
  y_values = [x**2 for x in x_values]

v plt.scatter(x_values, y_values, s=40)

  # Define o título do gráfico e nomeia os eixos
  --trecho omitido--

  # Define o intervalo para cada eixo
w plt.axis([0, 1100, 0, 1100000])

  plt.show()
```

Começamos com uma lista de valores de x contendo os números de 1 a 1.000 u. Em seguida, uma list comprehension gera os valores de y percorrendo os valores de x em um laço (`for x in x_values`), elevando cada número ao quadrado (`x**2`) e armazenando os resultados em `y_values`. Então passamos as listas de entrada e de saída para `scatter()` v.

Como esse é um conjunto bem grande de dados, usamos um tamanho menor de ponto e utilizamos a função `axis()` para especificar o intervalo de cada eixo w. A função `axis()` exige quatro valores: os valores mínimo e máximo para o eixo x e para o eixo y. Nesse caso, o eixo x varia de 0 a 1.100, e o eixo y, de 0 a 1.100.000. A Figura 15.6 mostra o resultado.

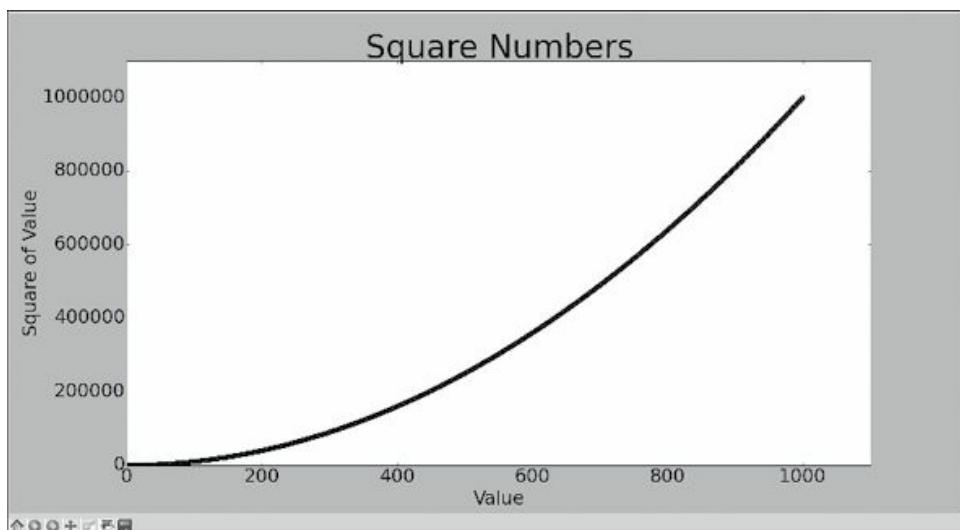



Figura 15.6 – Python é capaz de plotar 1.000 pontos tão facilmente quanto plota 5 pontos.

Removendo os contornos dos pontos de dados

O matplotlib permite colorir os pontos individualmente em um gráfico de dispersão. O padrão – pontos azuis com um contorno preto – funciona bem para gráficos com poucos pontos. Porém, ao plotar vários pontos, os contornos pretos podem se misturar. Para remover os contornos dos pontos, passe o argumento `edgecolor='none'` quando chamar `scatter()`:

```
plt.scatter(x_values, y_values, edgecolor='none', s=40)
```

Execute *scatter_squares.py* usando essa chamada e você deverá ver apenas pontos azuis sólidos em seu gráfico.

Definindo cores personalizadas

Para mudar a cor dos pontos, passe `c` para `scatter()` com o nome de uma cor a ser usada, como vemos a seguir:

```
plt.scatter(x_values, y_values, c='red', edgecolor='none', s=40)
```

Você também pode definir cores personalizadas usando o modelo de cores RGB. Para definir uma cor, passe uma tupla para o argumento `c`, com três valores decimais (um valor para cada cor, isto é, para vermelho, verde e azul), utilizando valores entre 0 e 1. Por exemplo, a linha a seguir cria um gráfico com pontos azuis claros:

```
plt.scatter(x_values, y_values, c=(0, 0, 0.8), edgecolor='none', s=40)
```

Valores mais próximos de 0 geram cores escuras enquanto valores mais próximos de 1 produzem cores mais claras.

Usando um colormap

Um *colormap* é uma série de cores em um gradiente que varia de uma cor inicial até uma cor final. Os colormaps são usados em visualizações para enfatizar um padrão nos dados. Por exemplo, você pode deixar os valores menores com uma cor clara e os valores maiores com uma cor mais escura.

O módulo `pyplot` inclui um conjunto de colormaps embutidos. Para usar um desses colormaps, especifique de que modo o `pyplot` deve atribuir uma

cor para cada ponto do conjunto de dados. Eis o modo de atribuir uma cor a cada ponto de acordo com o seu valor de y:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = list(range(1001))
y_values = [x**2 for x in x_values]

plt.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues,
    edgecolor='none', s=40)

# Define o título do gráfico e nomeia os eixos
--trecho omitido--
```

Passamos a lista de valores de y para `c` e, em seguida, informamos ao `pyplot` qual é o colormap a ser usado por meio do argumento `cmap`. Esse código pinta os pontos com valores menores de y com azul claro e os pontos com valores maiores de y com azul escuro. O gráfico resultante pode ser visto na Figura 15.7.

NOTA Você pode ver todos os colormaps disponíveis em `pyplot` em *http://matplotlib.org/*; acesse **Examples** (Exemplos), faça rolagens para baixo até Color Examples (Exemplos de cores) e clique em **colormaps_reference**.

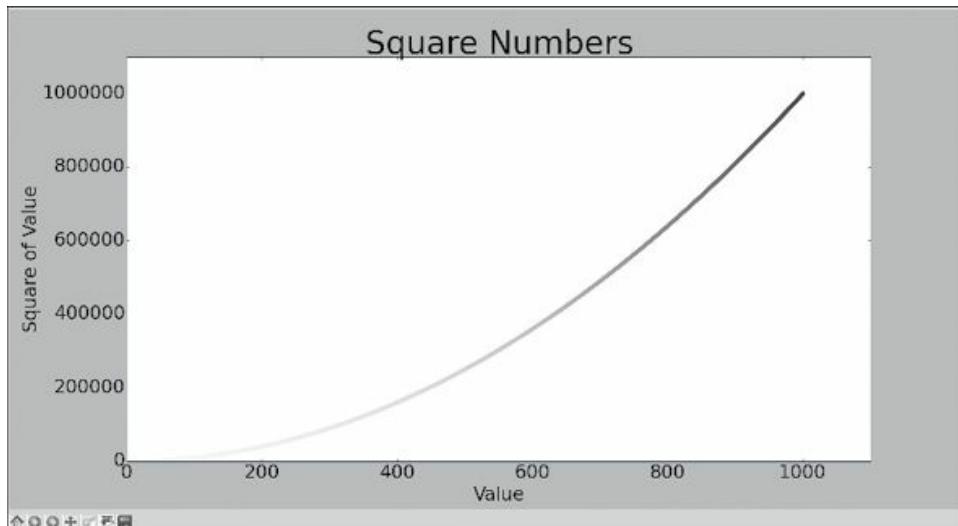



Figura 15.7 – Um gráfico que usa o colormap `Blues`.

Salvando seus gráficos automaticamente

Se quiser que seu programa salve automaticamente o gráfico em um arquivo, você poderá substituir a chamada a `plt.show()` por uma chamada a `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

O primeiro argumento é o nome de um arquivo para a imagem do gráfico, que será salvo no mesmo diretório que *scatter_squares.py*. O segundo argumento remove espaços em branco extras do gráfico. Se quiser ter espaços em branco extras ao redor do gráfico, você poderá omitir esse argumento.

FAÇA VOCÊ MESMO

15.1 – Cubos: Um número elevado à terceira potência é chamado de *cubo*. Faça a plotagem dos cinco primeiros números elevados ao cubo e, em seguida, dos primeiros 5.000 números elevados ao cubo.

15.2 – Cubos coloridos: Aplique um colormap ao seu gráfico de cubos.

Passeios aleatórios

Nesta seção usaremos Python para gerar dados a partir de um passeio aleatório e então usaremos o matplotlib para criar uma representação visualmente atraente dos dados gerados. Um *passeio aleatório* (random walk) é um caminho que não tem uma direção clara; ele é determinado por uma série de decisões aleatórias, em que cada uma é deixada totalmente ao acaso. Você pode imaginar um passeio aleatório como o caminho que uma formiga faria se tivesse enlouquecido e desse cada passo em uma direção aleatória.

Passeios aleatórios têm aplicações práticas na natureza, em física, biologia, química e economia. Por exemplo, um grão de pólen flutuando sobre uma gota d'água se move pela superfície porque é constantemente empurrada pelas moléculas de água. O movimento molecular em uma gota d'água é aleatório, portanto o caminho traçado por um grão de pólen na superfície é um passeio aleatório. O código que estamos prestes a escrever

modela muitas situações do mundo real.

Criando a classe RandomWalk()

Para implementar um passeio aleatório, criaremos uma classe `RandomWalk`, que tomará decisões aleatórias sobre a direção que o passeio deve seguir. A classe precisa de três atributos: uma variável para armazenar o número de pontos do passeio e duas listas para armazenar os valores das coordenadas x e y de cada ponto do passeio.

Usaremos apenas dois métodos na classe `RandomWalk`: o método `__init__()` e `fill_walk()`, que calculará os pontos do passeio. Vamos começar por `__init__()`, mostrado a seguir:

random_walk.py

```
u from random import choice

  class RandomWalk():
      """Uma classe para gerar passeios aleatórios."""

v     def __init__(self, num_points=5000):
          """Inicializa os atributos de um passeio."""
          self.num_points = num_points

          # Todos os passeios começam em (0, 0)
w          self.x_values = [0]
          self.y_values = [0]
```

Para tomar decisões aleatórias, armazenaremos possíveis opções em uma lista e usaremos `choice()` para decidir qual opção utilizaremos sempre que uma decisão for tomada u. Então definimos o número default de pontos de um passeio para 5.000 – um valor alto o suficiente para gerar alguns padrões interessantes, mas convenientemente baixo para gerar passeios de modo rápido v. Em seguida, em w, criamos duas listas para armazenar os valores de x e de y e começamos cada passeio no ponto (0, 0).

Escolhendo as direções

Usaremos `fill_walk()`, como mostrado a seguir, para preencher nosso

passeio com pontos e determinar a direção de cada passo. Adicione este método em *random_walk.py*:

random_walk.py

```
    def fill_walk(self):
        """Calcula todos os pontos do passeio."""

        # Continua dando passos até que o passeio alcance o tamanho
  desejado
u        while len(self.x_values) < self.num_points:

            # Decide direção a ser seguida e distância a ser percorrida
  nessa direção
v            x_direction = choice([1, -1])
            x_distance = choice([0, 1, 2, 3, 4])
w            x_step = x_direction * x_distance

            y_direction = choice([1, -1])
            y_distance = choice([0, 1, 2, 3, 4])
x            y_step = y_direction * y_distance

            # Rejeita movimentos que não vão a lugar nenhum
y            if x_step == 0 and y_step == 0:
                continue

            # Calcula os próximos valores de x e de y
z            next_x = self.x_values[-1] + x_step
            next_y = self.y_values[-1] + y_step

            self.x_values.append(next_x)
            self.y_values.append(next_y)
```

Em u criamos um laço que executa até o passeio ter sido preenchido com o número correto de pontos. A parte principal desse método diz a Python como simular quatro decisões aleatórias: O passeio seguirá para a direita ou para a esquerda? Qual a distância a ser percorrida nessa direção? O passeio seguirá para cima ou para baixo? Qual a distância a ser percorrida nessa direção?

Usamos `choice([1, -1])` para escolher um valor para `x_direction`, que

devolve 1 para um movimento à direita ou -1 para a esquerda v. Em seguida, `choice([0, 1, 2, 3, 4])` diz a Python qual é a distância a ser percorrida nessa direção (`x_distance`) ao selecionar aleatoriamente um inteiro entre 0 e 4. (A inclusão de 0 nos permite dar passos ao longo do eixo y, além dos passos com movimentos ao longo dos dois eixos.)

Em w e x determinamos o tamanho de cada passo nas direções *x* e *y* multiplicando a direção do movimento pela distância escolhida. Um resultado positivo para `x_step` nos move para a direita, um resultado negativo nos move para a esquerda e 0 nos move verticalmente. Um resultado positivo para `y_step` significa um movimento para cima, um valor negativo representa um movimento para baixo e 0 implica um movimento na horizontal. Se o valor tanto de `x_step` quanto de `y_step` forem 0, o passeio é interrompido, mas continuamos o laço para evitar isso y.

Para obter o próximo valor de x de nosso passeio, somamos o valor em `x_step` ao último valor armazenado em `x_values` z e fazemos o mesmo para os valores de y. Depois que tivermos esses valores, eles são concatenados em `x_values` e em `y_values`.

Plotando o passeio aleatório

Eis o código para plotar todos os pontos do passeio:

rw_visual.py

```
  import matplotlib.pyplot as plt

  from random_walk import RandomWalk

  # Cria um passeio aleatório e plota os pontos
u rw = RandomWalk()
  rw.fill_walk()
v plt.scatter(rw.x_values, rw.y_values, s=15)
  plt.show()
```

Começamos importando `pyplot` e `RandomWalk`. Então criamos um passeio aleatório, armazenamos esse passeio em `rw` u e garantimos que `fill_walk()` seja chamado. Em v passamos os valores de x e de y do passeio para `scatter()` e escolhemos um tamanho apropriado para o ponto. A Figura

15.8 mostra o gráfico resultante com 5.000 pontos. (As imagens desta seção omitem o visualizador do matplotlib, mas você continuará a vê-lo quando executar *rw_visual.py*.)

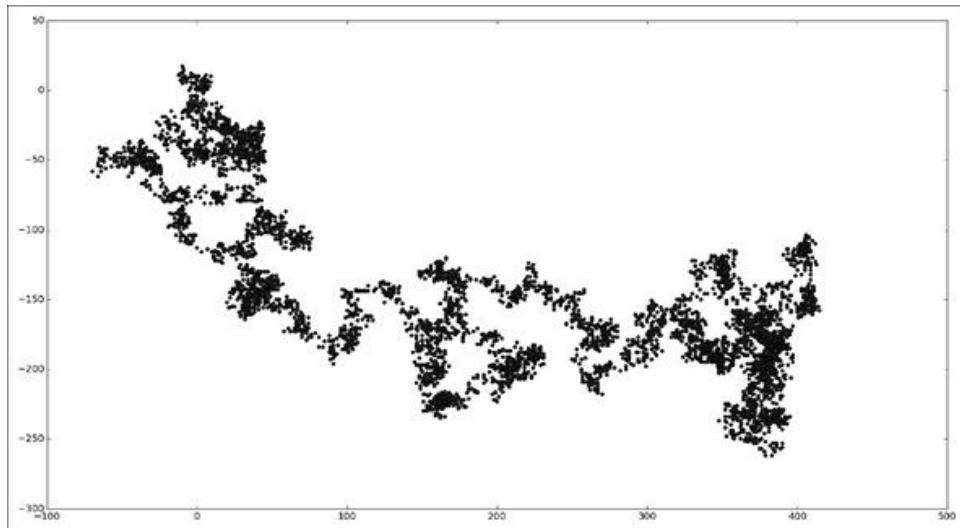



Figura 15.8 – Um passeio aleatório com 5.000 pontos.

Gerando vários passeios aleatórios

Todo passeio aleatório é diferente, e é divertido explorar os vários padrões que podem ser gerados. Uma maneira de usar o código anterior para gerar vários passeios sem a necessidade de executar o programa diversas vezes é colocá-lo em um laço `while`, assim:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Continua criando novos passeios enquanto o programa estiver ativo
while True:
    # Cria um passeio aleatório e plota os pontos
    rw = RandomWalk()
    rw.fill_walk()
    plt.scatter(rw.x_values, rw.y_values, s=15)
    plt.show()

u     keep_running = input("Make another walk? (y/n): ")
```

```
    if keep_running == 'n':
        break
```

Esse código gera um passeio aleatório, exibe esse passeio no visualizador do matplotlib e faz uma pausa com o visualizador aberto. Quando você o fechar, uma pergunta será feita para saber se você quer gerar outro passeio. Responda **y** e você poderá gerar passeios que permaneçam próximos ao ponto de partida, que se afastam em uma direção principal, que têm seções mais finas conectadas a grupos maiores de pontos, e assim por diante. Quando quiser encerrar o programa, digite **n**.

NOTA Se você usa Python 2.7, lembre-se de utilizar `raw_input()` no lugar de `input()` em **u**.

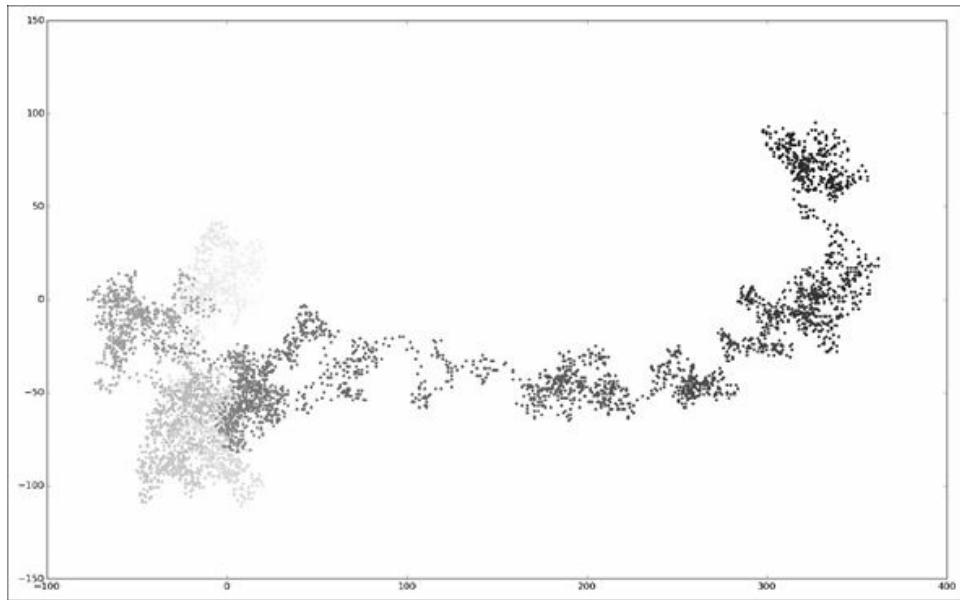
Estilizando o passeio

Nesta seção personalizaremos nossos gráficos para enfatizar as características importantes de cada passeio e remover a ênfase dos elementos que causam distração. Para isso, identificamos as características que queremos enfatizar, por exemplo, o ponto de partida do passeio, seu ponto final e o percurso feito. Em seguida, identificamos as características que não serão enfatizadas, como as marcações e os rótulos. O resultado deve ser uma representação visual simples que comunique claramente o caminho tomado em cada passeio aleatório.

Colorindo os pontos

Usaremos um colormap para mostrar a ordem dos pontos no passeio e então removeremos o contorno preto de cada ponto para que a cor deles seja mais evidente. Para colorir os pontos de acordo com suas posições no passeio, passamos uma lista contendo a posição de cada ponto no argumento `c`. Como os pontos são plotados na sequência, a lista simplesmente contém os números de 1 a 5.000, como vemos a seguir:

rw_visual.py

```
--trecho omitido--
while True:
    # Cria um passeio aleatório e plota os pontos
```

```
    rw = RandomWalk()
    rw.fill_walk()

u   point_numbers = list(range(rw.num_points))
    plt.scatter(rw.x_values, rw.y_values, c=point_numbers,
  cmap=plt.cm.Blues,
        edgecolor='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    --trecho omitido--
```

Em u usamos `range()` para gerar uma lista de números igual ao número de pontos do passeio. Então armazenamos esses números na lista `point_numbers`, que usaremos para definir a cor de cada ponto do passeio. Passamos `point_numbers` para o argumento `c`, usamos o colormap `Blues` e passamos `edgecolor=none` para eliminar o contorno preto de cada ponto. O resultado é um gráfico do passeio que varia do azul claro para o azul escuro ao longo de um gradiente, como vemos na Figura 15.9.

Figura 15.9 – Um passeio aleatório colorido com o colormap `Blues`.

Plotando os pontos de início e de fim

Além de colorir os pontos para mostrar as suas posições no decorrer do passeio, seria interessante ver em que lugar cada passeio começa e termina.

Para isso, podemos plotar o primeiro e o último ponto individualmente depois que a série principal for plotada. Deixaremos os pontos de início e de fim maiores e usaremos cores diferentes para destacá-los, como vemos a seguir:

rw_visual.py

```
--trecho omitido--
while True:
    --trecho omitido--
    plt.scatter(rw.x_values, rw.y_values, c=point_numbers,
cmap=plt.cm.Blues,
        edgecolor='none', s=15)

    # Enfatiza o primeiro e o último ponto
    plt.scatter(0, 0, c='green', edgecolors='none', s=100)
    plt.scatter(rw.x_values[-1], rw.y_values[-1], c='red',
edgecolors='none',
        s=100)

    plt.show()
    --trecho omitido--
```

Para mostrar o ponto inicial, plotamos o ponto (0, 0) em verde com um tamanho maior (`s=100`) que o restante dos pontos. Para marcar o ponto final, plotamos o último valor de x e y do passeio em vermelho, com um tamanho igual a 100. Lembre-se de inserir esse código logo antes da chamada a `plt.show()` para que os pontos de início e de fim sejam desenhados sobre todos os demais pontos.

Ao executar esse código, você deverá ser capaz de identificar exatamente em que ponto cada passeio começa e termina. (Se esses pontos de início e de fim não estiverem claramente em destaque, ajuste suas cores e tamanhos até isso acontecer.)

Limpando os eixos

Vamos remover os eixos desse gráfico para que eles não nos distraiam do caminho de cada passeio. Para desabilitar os eixos, utilize este código:

rw_visual.py

```
--trecho omitido--
while True:
    --trecho omitido--
    plt.scatter(rw.x_values[-1], rw.y_values[-1], c='red',
edgecolors='none',
        s=100)

    # Remove os eixos
u     plt.axes().get_xaxis().set_visible(False)
    plt.axes().get_yaxis().set_visible(False)

    plt.show()
    --trecho omitido--
```

Para modificar os eixos, use a função `plt.axes()` u para definir a visibilidade de cada um deles com `False`. À medida que continuar a trabalhar com visualizações, você verá essa cadeia de métodos com frequência.

Execute *rw_visual.py* agora; você deverá ver uma série de plotagens sem eixos.

Adicionando pontos para plotagem

Vamos aumentar o número de pontos para termos mais dados com os quais possamos trabalhar. Para isso, aumentamos o valor de `num_points` na criação de uma instância de `RandomWalk` e ajustamos o tamanho de cada ponto quando desenhamos o gráfico, como vemos a seguir:

rw_visual.py

```
--trecho omitido--
while True:
    # Cria um passeio aleatório e plota os pontos
    rw = RandomWalk(50000)
    rw.fill_walk()

    # Plota os pontos e mostra o gráfico
    point_numbers = list(range(rw.num_points))
```

```
    plt.scatter(rw.x_values, rw.y_values, c=point_numbers,
cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    --trecho omitido--
```

Esse exemplo cria um passeio aleatório com 50.000 pontos (para espelhar dados do mundo real) e plota cada ponto com um tamanho `s=1`. O passeio resultante tem um aspecto etéreo e lembra uma nuvem, como mostrado na Figura 15.10. Como podemos ver, criamos uma obra de arte a partir de um simples gráfico de dispersão!

Faça experimentos com esse código para ver até que ponto você pode aumentar o número de pontos de um passeio antes que seu sistema comece a ficar lento de forma significativa ou o gráfico deixe de ser visualmente atraente.

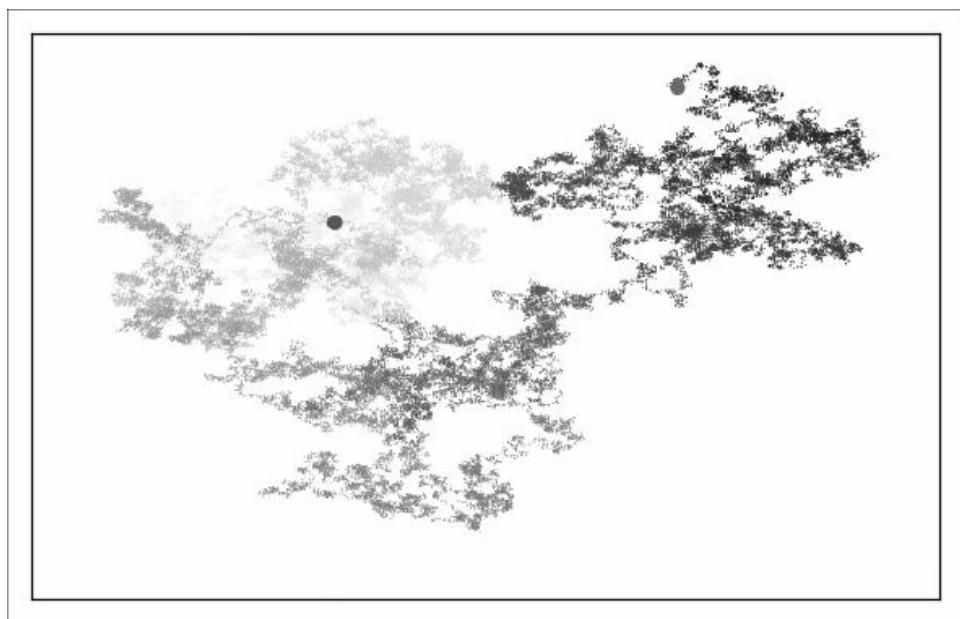



Figura 15.10 – Um passeio aleatório com 50.000 pontos.

Alterando o tamanho para preencher a tela

Uma visualização será muito mais eficaz para informar padrões em dados se couber apropriadamente na tela. Para deixar a janela de plotagem mais adequada à sua tela, ajuste o tamanho da saída do matplotlib, assim:

rw_visual.py

```
--trecho omitido--
while True:
    # Cria um passeio aleatório e plota os pontos
    rw = RandomWalk()
    rw.fill_walk()

    # Define o tamanho da janela de plotagem
    plt.figure(figsize=(10, 6))
    --trecho omitido--
```

A função `figure()` controla a largura, a altura, a resolução e a cor de fundo do gráfico. O parâmetro `figsize` aceita uma tupla, que informa ao matplotlib as dimensões da janela de plotagem em polegadas.

Python pressupõe que a resolução de sua tela seja de 80 pixels por polegada; se esse código não oferecer um tamanho exato para o seu gráfico, ajuste os números conforme for necessário. Por outro lado, se você souber qual é a resolução de seu sistema, passe-a para `figure()` usando o parâmetro `dpi` para definir um tamanho de gráfico que faça um uso eficiente do espaço disponível em sua tela, como vemos a seguir:

```
plt.figure(dpi=128, figsize=(10, 6))
```

FAÇA VOCÊ MESMO

15.3 – Movimento molecular: Modifique *rw_visual.py* substituindo `plt.scatter()` por `plt.plot()`. Para simular o percurso de um grão de pólen na superfície de uma gota d'água, passe `rw.x_values` e `rw.y_values` e inclua um argumento `linewidth`. Utilize 5.000 em vez de 50.000 pontos.

15.4 – Passeios aleatórios modificados: Na classe `RandomWalk`, `x_step` e `y_step` são gerados a partir do mesmo conjunto de condições. A direção é escolhida aleatoriamente a partir da lista [1, -1], e a distância, a partir da lista [0, 1, 2, 3, 4]. Modifique os valores dessas listas para ver o que acontece com o formato geral de seus passeios. Experimente usar uma lista maior de opções para a distância, por exemplo, de 0 a 8, ou remova o -1 da lista de direção *x* ou *y*.

15.5 – Refatoração: O método `fill_walk()` é longo. Crie um novo método chamado `get_step()` para determinar a direção e a distância de cada passo, e depois calcule o passo. Você deverá ter duas chamadas para `get_step()` em `fill_walk()`:

```
x_step = get_step()
y_step = get_step()
```

Essa refatoração deverá reduzir o tamanho de `fill_walk()` e deixar o método mais fácil de ler e de entender.

Lançando dados com o Pygal

Nesta seção usaremos o pacote de visualização Pygal de Python para produzir arquivos com gráficos vetoriais escaláveis. Eles são úteis em visualizações apresentadas em telas de tamanhos diferentes, pois são automaticamente escaladas para se adequar à tela de quem as estiver vendo. Se você planeja usar suas visualizações online, considere usar o Pygal para que seu trabalho tenha uma boa aparência em qualquer dispositivo usado pelas pessoas para as visualizações.

Neste projeto analisaremos os resultados do lançamento de dados. Se você lançar um dado comum de seis lados, terá a mesma chance de obter qualquer um dos números de 1 a 6. No entanto, ao usar dois dados, é mais provável que você tire determinados números e não outros. Tentaremos determinar quais números são mais prováveis de ocorrer gerando um conjunto de dados que represente o lançamento dos dados. Então plotaremos o resultado de um número maior de lançamentos para determinar quais resultados são mais prováveis que outros.

O estudo do lançamento de dados muitas vezes é usado em matemática para explicar vários tipos de análise de dados. No entanto, esses estudos também têm aplicações no mundo real, em cassinos e em outros cenários de apostas, assim como na forma como se comportam jogos como o Monopólio e jogos de RPG (Role-Playing Game).

Instalando o Pygal

Instale o Pygal usando `pip`. (Se você ainda não usou o `pip`, consulte a seção “Instalando pacotes Python com o pip”.)

No Linux e no OS X, esse comando deve ser semelhante a:

```
pip install --user pygal
```

No Windows, deve ser:

```
python -m pip install --user pygal
```

NOTA Talvez você precise usar o comando `pip3` em vez de `pip`; se o comando ainda não funcionar, pode ser que seja necessário remover a flag `--user`.

Galeria do Pygal

Para ver os tipos de visualizações possíveis com o Pygal, visite a galeria de tipos de gráficos: acesse *http://www.pygal.org/*, clique em **Documentation** (Documentação) e depois em **Chart types** (Tipos de gráfico). Todo exemplo inclui o código-fonte, portanto você poderá ver como as visualizações são geradas.

Criando a classe Die

Eis uma classe para simular o lançamento de um dado:

die.py

```
  from random import randint

  class Die():
      """Uma classe que representa um único dado."""

u     def __init__(self, num_sides=6):
          """Supõe que seja um dado de seis lados."""
          self.num_sides = num_sides

      def roll(self):
          """"Devolve um valor aleatório entre 1 e o número de lados."""
v         return randint(1, self.num_sides)
```

O método `__init__()` aceita um argumento opcional. Com essa classe, quando uma instância de nosso dado for criada, o número de lados sempre será seis se nenhum argumento for incluído. Se um argumento *for* incluído, esse valor será usado para definir o número de lados do dado u. (Os dados são nomeados de acordo com o seu número de lados: um dado de seis lados é um D6, um dado de oito lados é um D8, e assim por diante.)

O método roll() usa a função randint() para devolver um número aleatório entre 1 e o número de lados v. Essa função pode devolver o valor inicial (1), o valor final (num_sides) ou qualquer inteiro entre eles.

Lançando o dado

Antes de criar uma visualização baseada nessa classe, vamos lançar um D6, exibir o resultado e verificar se ele parece razoável:

die_visual.py

```
  from die import Die

  # Cria um D6
u die = Die()

  # Faz alguns lançamentos e armazena os resultados em uma lista
  results = []
v for roll_num in range(100):
      result = die.roll()
      results.append(result)

  print(results)
```

Em u criamos uma instância de Die com seis lados como default. Em v lançamos o dado cem vezes e armazenamos o resultado de cada lançamento na lista results. Eis uma amostra do conjunto de resultados:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5,
4,
1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
1, 5, 1, 2]
```

Uma observação rápida desse resultado mostra que a classe Die parece estar funcionando. Podemos ver os valores de 1 a 6, portanto sabemos que o menor e o maior valores possíveis são devolvidos; como não vemos 0 nem 7, sabemos que todos os resultados estão no intervalo apropriado. Também vemos todos os números de 1 a 6, o que indica que todos os

resultados possíveis estão representados.

Analisando os resultados

Analisamos os resultados do lançamento de um D6 contando quantas vezes tiramos cada número:

die_visual.py

```
  --trecho omitido--
  # Faz alguns lançamentos e armazena os resultados em uma lista
  results = []
u for roll_num in range(1000):
      result = die.roll()
      results.append(result)

  # Analisa os resultados
  frequencies = []
v for value in range(1, die.num_sides+1):
w     frequency = results.count(value)
x     frequencies.append(frequency)

  print(frequencies)
```

Como estamos usando o Pygal para analisar e não para exibir os resultados, podemos aumentar o número de lançamentos simulados para 1.000 u. Para analisar os lançamentos, criamos a lista vazia `frequencies` para armazenar o número de vezes que cada valor foi tirado. Percorremos os valores possíveis com um laço (de 1 a 6, nesse caso) em v, contamos quantas vezes cada número aparece em `results` w e concatenamos esse valor na lista `frequencies` x. Então exibimos essa lista antes de criar uma visualização:

```
[155, 167, 168, 170, 159, 181]
```

Esse resultado parece ser razoável: vemos seis frequências, uma para cada número possível quando lançamos um D6, e vemos que nenhuma frequência é significativamente maior que as demais. Vamos agora visualizar esses resultados.

Criando um histograma

Com uma lista de frequências, podemos criar um histograma dos resultados. Um *histograma* é um gráfico de barras que mostra a frequência da ocorrência de determinados resultados. Eis o código para criar o histograma:

die_visual.py

```
  import pygal
  --trecho omitido--

  # Analisa os resultados
  frequencies = []
  for value in range(1, die.num_sides+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Visualiza os resultados
u hist = pygal.Bar()

  hist.title = "Results of rolling one D6 1000 times."
v hist.x_labels = ['1', '2', '3', '4', '5', '6']
  hist.x_title = "Result"
  hist.y_title = "Frequency of Result"

w hist.add('D6', frequencies)
  hist.render_to_file('die_visual.svg')
```

Geramos um gráfico de barras criando uma instância de `pygal.Bar()`, que armazenamos em `hist` **u**. Então definimos o atributo `title` de `hist` (apenas uma string que usamos para dar nome ao histograma), usamos os resultados possíveis do lançamento de um D6 como rótulos no eixo x **v** e adicionamos um título para cada um dos eixos. Usamos `add()` para acrescentar uma série de valores ao gráfico em **w** (passando-lhe um rótulo para o conjunto de valores a ser adicionado e uma lista dos valores que aparecerão no gráfico). Por fim, renderizamos o gráfico em um arquivo SVG, que espera um nome de arquivo com a extensão *.svg*.

A maneira mais simples de ver o histograma resultante é usar um

navegador web. Crie uma nova aba em qualquer navegador web e abra o arquivo *die_visual.svg* (na pasta em que você salvou *die_visual.py*). Você deverá ver um gráfico como o que está na Figura 15.11. (Modifiquei um pouco esse gráfico para a exibição; por padrão, o Pygal gera gráficos com uma cor de fundo mais escura que essa que você vê aqui.)

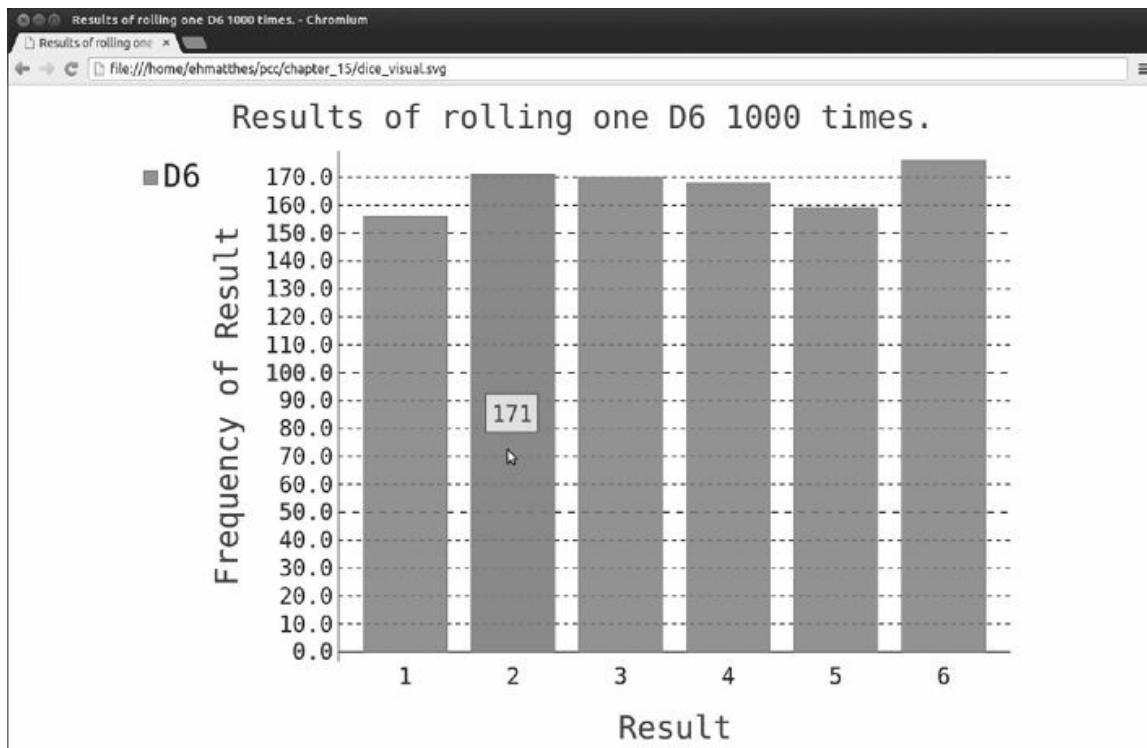



Figura 15.11 – Um gráfico de barras simples criado com o Pygal.

Observe que o Pygal fez o gráfico ser interativo: passe o cursor sobre qualquer barra do gráfico e você verá o dado associado a ela. Esse recurso é particularmente útil quando plotamos vários conjuntos de dados no mesmo gráfico.

Lançando dois dados

Lançar dois dados resulta em números maiores e uma distribuição diferente dos resultados. Vamos modificar o nosso código para criar dois dados D6 a fim de simular o modo como lançamos um par de dados. Sempre que lançarmos o par, somaremos os dois números (um de cada dado) e armazenaremos o resultado em `results`. Salve uma cópia de *die_visual.py* como *dice_visual.py* e faça as seguintes alterações:

dice_visual.py

```
  import pygal

  from die import Die

  # Cria dois dados D6
  die_1 = Die()
  die_2 = Die()

  # Faz alguns lançamentos e armazena os resultados em uma lista
  results = []
  for roll_num in range(1000):
u     result = die_1.roll() + die_2.roll()
      results.append(result)

  # Analisa os resultados
  frequencies = []
v max_result = die_1.num_sides + die_2.num_sides
w for value in range(2, max_result+1):
      frequency = results.count(value)
      frequencies.append(frequency)

  # Visualiza os resultados
  hist = pygal.Bar()

x hist.title = "Results of rolling two D6 dice 1000 times."
  hist.x_labels = ['2', '3', '4', '5', '6', '7', '8', '9', '10', '11',
  '12']
  hist.x_title = "Result"
  hist.y_title = "Frequency of Result"

  hist.add('D6 + D6', frequencies)
  hist.render_to_file('dice_visual.svg')
```

Depois de criar duas instâncias de `Die`, lançamos os dados e calculamos a soma dos dois dados em cada lançamento u. O maior resultado possível (12) é a soma do maior número nos dois dados, e nós o armazenamos em `max_result` v. O menor resultado possível (2) é a soma do menor número nos dois dados. Quando analisamos os resultados, contamos o número de

resultados para cada valor entre 2 e `max_result` W. (Poderíamos ter usado `range(2, 13)`, mas isso funcionaria somente para dois dados D6. Ao modelar situações do mundo real, é melhor escrever um código que possa modelar facilmente várias situações. Esse código nos permite simular o lançamento de um par de dados com qualquer quantidade de lados.)

Quando criamos o gráfico, atualizamos o título e os rótulos do eixo x e da série de dados X. (Se a lista `x_labels` for muito longa, faria sentido escrever um laço para gerar essa lista automaticamente.)

Depois de executar esse código, atualize a aba que mostra o gráfico em seu navegador: você deverá ver um gráfico como o que está na Figura 15.12.

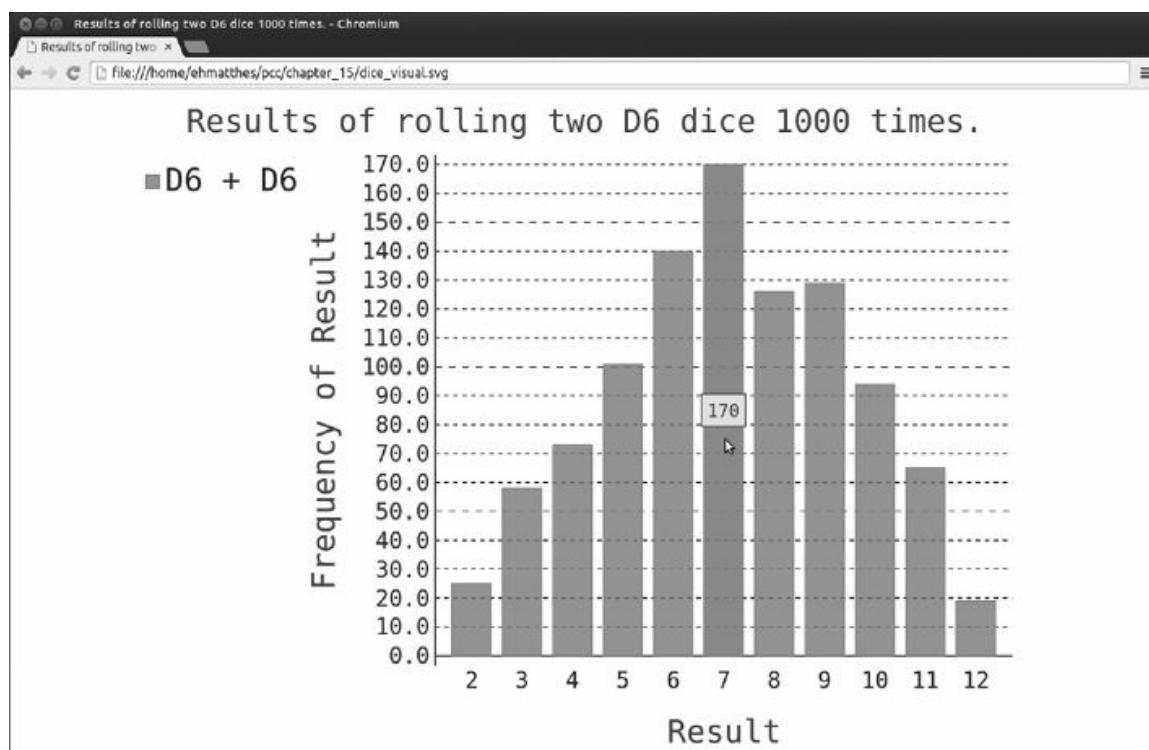



Figura 15.12 – Resultados simulados para o lançamento de dois dados de seis lados, 1.000 vezes.

Esse gráfico mostra os resultados aproximados que você provavelmente obterá quando lançar um par de dados D6. Como podemos ver, é menos provável que você obtenha um 2 ou um 12, e mais provável que tire um 7, pois há seis maneiras de obter esse valor: 1 e 6, 2 e 5, 3 e 4, 4 e 3, 5 e 2 ou 6 e 1.

Lançando dados de tamanhos diferentes

Vamos criar um dado de seis lados e outro de dez lados e ver o que acontece quando os lançamos 50.000 vezes.

different_dice.py

```
  from die import Die

  import pygal

  # Cria um D6 e um D10
  die_1 = Die()
u die_2 = Die(10)

  # Faz alguns lançamentos e armazena os resultados em uma lista
  results = []
  for roll_num in range(50000):
      result = die_1.roll() + die_2.roll()
      results.append(result)

  # Analisa os resultados
  --trecho omitido--

  # Visualiza os resultados
  hist = pygal.Bar()

v hist.title = "Results of rolling a D6 and a D10 50,000 times."
  hist.x_labels = ['2', '3', '4', '5', '6', '7', '8', '9', '10', '11',
  '12',
      '13', '14', '15', '16']
  hist.x_title = "Result"
  hist.y_title = "Frequency of Result"

  hist.add('D6 + D10', frequencies)
  hist.render_to_file('dice_visual.svg')
```

Para criar um D10, passamos o argumento `10` na criação da segunda instância de `Die` u e mudamos o primeiro laço para simular 50.000 lançamentos em vez de 1.000. O menor resultado possível ainda é 2, mas o maior resultado agora é 16; desse modo, ajustamos o título, os rótulos do

eixo x e os rótulos das séries de dados para refletir esse fato **v**.

A Figura 15.13 mostra o gráfico resultante. Em vez de haver um único resultado mais provável, temos cinco. Isso acontece porque continua havendo apenas uma maneira de obter o menor valor (1 e 1) e o maior valor (6 e 10), porém o dado menor limita o número de maneiras pelas quais podemos gerar os números intermediários: há seis maneiras de obter os resultados 7, 8, 9, 10 e 11. Portanto esses são os resultados mais comuns e é igualmente provável que você obtenha qualquer um desses números.

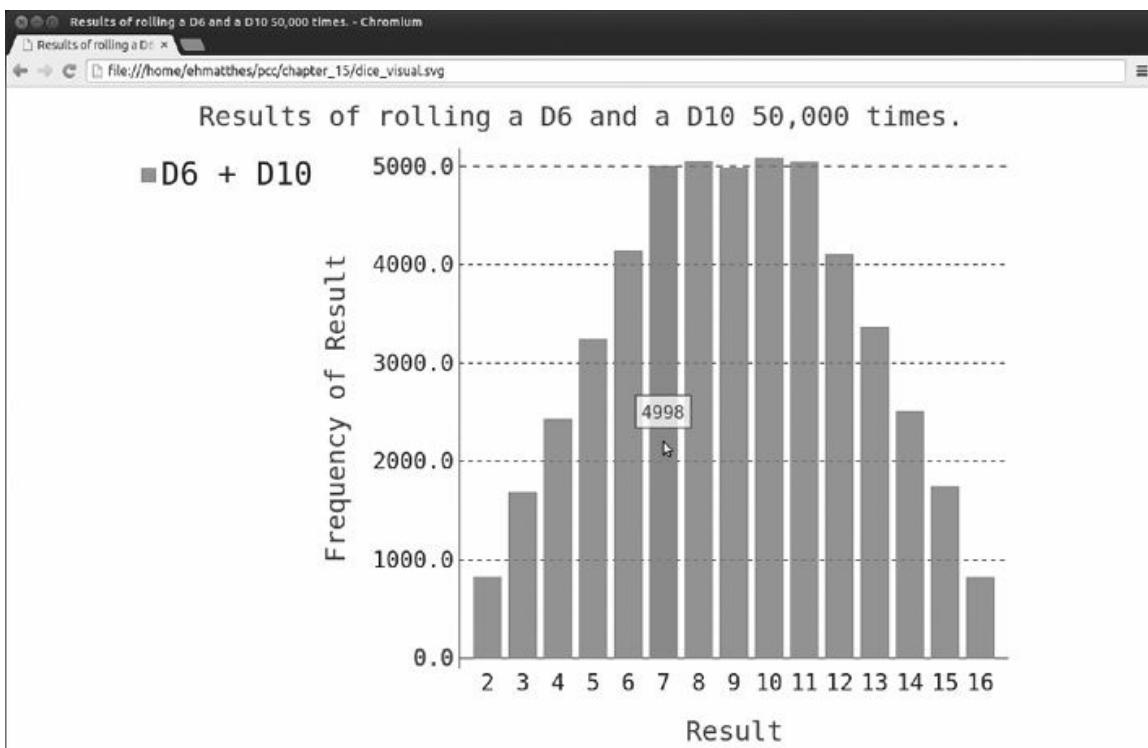



Figura 15.13 – Os resultados do lançamento de um dado de seis lados e outro de dez lados, 50.000 vezes.

Nossa capacidade de usar o Pygal para modelar o lançamento de dados nos proporciona uma liberdade considerável para explorar esse fenômeno. Em apenas alguns minutos, você pode simular um número excepcional de lançamentos usando uma grande variedade de dados.

FAÇA VOCÊ MESMO

15.6 – Rótulos automáticos: Modifique *die.py* e *dice_visual.py* substituindo a lista que usamos para definir o valor de `hist.x_labels` por um laço que gere essa lista

automaticamente. Se você se sentir à vontade com as list comprehensions, experimente substituir os outros laços `for` em *die_visual.py* e em *dice_visual.py* por comprehensions também.

15.7 – Dois D8s: Crie uma simulação que mostre o que acontece se você lançar dois dados de oito lados, 1.000 vezes. Aumente o número de lançamentos gradualmente até começar a perceber os limites da capacidade de seu sistema.

15.8 – Três dados: Se lançar três dados D6, o menor número que você poderá obter é 3 e o maior número é 18. Crie uma visualização que mostre o que acontece quando lançamos três dados D6.

15.9 – Multiplicação: Quando lançamos dois dados, geralmente somamos os números para obter o resultado. Crie uma visualização que mostre o que acontece se você multiplicar esses números em vez de somá-los.

15.10 – Exercitando as duas bibliotecas: Experimente usar o matplotlib para criar uma visualização de lançamento de dados e use o Pygal para criar uma visualização de um passeio aleatório.

Resumo

Neste capítulo aprendemos a gerar conjuntos de dados e a criar visualizações desses dados. Vimos como criar gráficos simples com o matplotlib e usar um gráfico de dispersão para explorar passeios aleatórios. Aprendemos a criar um histograma com o Pygal e vimos como usar um histograma para explorar os resultados do lançamento de dados de diferentes tamanhos.

Gerar seus próprios conjuntos de dados usando código é uma maneira interessante e eficaz de modelar e de explorar uma grande variedade de situações do mundo real. À medida que continuar a trabalhar com os próximos projetos de visualização de dados, fique de olho em situações que você possa modelar usando código. Observe as visualizações que estão nas mídias de notícias e veja se você é capaz de identificar aquelas que foram geradas com métodos semelhantes aos que conhecemos nesses projetos.

No Capítulo 16 faremos o download de dados a partir de fontes online e continuaremos a usar o matplotlib e o Pygal para explorar esses dados.

16
FAZENDO DOWNLOAD DE DADOS

Neste capítulo você fará download de conjuntos de dados a partir de fontes online e criará visualizações funcionais desses dados. Uma grande variedade de dados pode ser encontrada online, e muitos desses ainda não foram analisados de forma completa. A capacidade de analisar esses dados permite descobrir padrões e conexões que ninguém mais percebeu.

Vamos acessar e visualizar dados armazenados em dois formatos comuns: CSV e JSON. Usaremos o módulo `csv` de Python para processar dados meteorológicos armazenados no formato CSV (Comma-Separated Values, ou Valores Separados por Vírgula) e analisar as temperaturas máxima e mínima ao longo do tempo em duas localidades diferentes. Então usaremos o matplotlib para gerar um gráfico com base nos dados baixados e exibiremos as variações de temperatura em dois ambientes bem distintos: Sitka no Alasca e o Vale da Morte (Death Valley) na Califórnia. Mais adiante, neste capítulo, usaremos o módulo `json` para acessar dados de população armazenados no formato JSON e usaremos o Pygal para desenhar um mapa da população de cada país.

No final deste capítulo você estará preparado para trabalhar com diferentes tipos e formatos de conjuntos de dados e compreenderá melhor

o processo de criação de visualizações complexas. A capacidade de acessar e de visualizar dados online de tipos e formatos diferentes é essencial para trabalhar com uma grande variedade de conjuntos de dados do mundo real.

Formato de arquivo CSV

Uma maneira simples de armazenar dados em um arquivo-texto é escrevê-los como uma série de *valores separados por vírgula* (comma-separated values). Os arquivos resultantes são chamados de arquivos *CSV*. Por exemplo, eis uma linha de dados meteorológicos no formato CSV:

```
2014-1-
5,61,44,26,18,7,-1,56,30,9,30.34,30.27,30.15,,,,10,4,,0.00,0,,195
```

São dados meteorológicos de 5 de janeiro de 2014 para Sitka no Alasca. Incluem as temperaturas máxima e mínima assim como várias outras medições desse dia. Os arquivos CSV podem ser complicados para os seres humanos lerem, mas são fáceis para os programas processarem e extraírem valores, o que agiliza a operação de análise de dados.

Começaremos com um pequeno conjunto de dados meteorológicos formatados em CSV, registrados em Sitka e disponíveis nos recursos do livro em *https://www.nostarch.com/pythoncrashcourse/*. Copie o arquivo *sitka_weather_07-2014.csv* para a pasta em que você está escrevendo os programas deste capítulo. (Depois que fizer o download dos recursos do livro, você terá todos os arquivos necessários a este projeto.)

NOTA Os dados meteorológicos deste projeto foram originalmente baixados de *http://www.wunderground.com/history/*.

Fazendo parse dos cabeçalhos de arquivos CSV

O módulo `csv` da biblioteca-padrão de Python faz parse das linhas de um arquivo CSV e permite extrair rapidamente os valores em que estivermos interessados. Vamos começar analisando a primeira linha do arquivo, que contém uma série de cabeçalhos para os dados:

highs_lows.py

```
  import csv

  filename = 'sitka_weather_07-2014.csv'
u with open(filename) as f:
v     reader = csv.reader(f)
w     header_row = next(reader)
      print(header_row)
```

Depois de importar o módulo `csv`, armazenamos o nome do arquivo com que estamos trabalhando em `filename`. Então abrimos o arquivo e armazenamos o objeto arquivo resultante em `f` **u**. Em seguida, chamamos `csv.reader()` e lhe passamos o objeto arquivo como argumento a fim de criar um objeto reader associado a esse arquivo **v**. Armazenamos o objeto reader em `reader`.

O módulo `csv` contém uma função `next()`, que devolve a próxima linha do arquivo quando recebe o objeto reader. Na listagem anterior, chamamos `next()` apenas uma vez para obter a primeira linha do arquivo, que contém os cabeçalhos **w**. Armazenamos os dados devolvidos em `header_row`. Como podemos ver, `header_row` contém cabeçalhos significativos relacionados a dados meteorológicos, que nos informam quais dados estão armazenados em cada linha:

```
['AKDT', 'Max TemperatureF', 'Mean TemperatureF', 'Min TemperatureF',
'Max Dew PointF', 'MeanDew PointF', 'Min DewpointF', 'Max Humidity',
' Mean Humidity', ' Min Humidity', ' Max Sea Level PressureIn',
' Mean Sea Level PressureIn', ' Min Sea Level PressureIn',
' Max VisibilityMiles', ' Mean VisibilityMiles', ' Min
VisibilityMiles',
' Max Wind SpeedMPH', ' Mean Wind SpeedMPH', ' Max Gust SpeedMPH',
'PrecipitationIn', ' CloudCover', ' Events', ' WindDirDegrees']
```

`reader` processa a primeira linha de valores separados por vírgula do arquivo e armazena cada um deles como um item em uma lista. O cabeçalho `AKDT` representa Alaska Daylight Time (Fuso horário do Alasca). A posição desse cabeçalho nos informa que o primeiro valor de cada linha será a data ou a hora. O cabeçalho `Max TemperatureF` nos informa que o segundo valor de cada linha é a temperatura máxima dessa data em graus Fahrenheit. Você pode ler o restante dos cabeçalhos para determinar quais

tipos de informação estão incluídos no arquivo.

NOTA Os cabeçalhos nem sempre estão formatados de modo consistente: espaços e unidades estão em lugares inusitados. Isso é comum em arquivos com dados brutos, mas não causarão problemas.

Exibindo os cabeçalhos e suas posições

Para facilitar a compreensão dos dados de cabeçalho do arquivo, exiba cada cabeçalho e a sua posição na lista:

highs_lows.py

```
--trecho omitido--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

u     for index, column_header in enumerate(header_row):
          print(index, column_header)
```

Usamos `enumerate()` u na lista para obter o índice de cada item, assim como o valor. (Observe que removemos a linha `print(header_row)` em troca dessa versão mais detalhada.)

Eis a saída mostrando o índice de cada cabeçalho:

```
0 AKDT
1 Max TemperatureF
2 Mean TemperatureF
3 Min TemperatureF
--trecho omitido--
20  CloudCover
21  Events
22  WindDirDegrees
```

Nessa saída vemos que as datas e suas temperaturas máximas estão armazenadas nas colunas 0 e 1. Para explorar esses dados, processaremos cada linha de *sitka_weather_07-2014.csv* e extrairemos os valores nos índices 0 e 1.

Extraindo e lendo dados

Agora que sabemos quais são as colunas de dados de que precisaremos, vamos ler alguns desses dados. Em primeiro lugar, vamos ler a temperatura máxima de cada dia:

highs_lows.py

```
  import csv

  # Obtém as temperaturas máximas do arquivo
  filename = 'sitka_weather_07-2014.csv'
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

u     highs = []
v     for row in reader:
w         highs.append(row[1])

      print(highs)
```

Criamos uma lista vazia chamada `highs` u e então percorremos as linhas restantes do arquivo com um laço v. O objeto reader continua a partir de onde parou no arquivo CSV e devolve automaticamente cada linha após a sua posição atual. Como já lemos a linha com o cabeçalho, o laço começará na segunda linha, em que os dados propriamente ditos têm início. A cada passagem pelo laço concatenamos os dados do índice 1 – a segunda coluna – em `highs` w.

A listagem a seguir mostra os dados que estão agora armazenados em `highs`:

```
['64', '71', '64', '59', '69', '62', '61', '55', '57', '61', '57',
'59', '57',
'61', '64', '61', '59', '63', '60', '57', '69', '63', '62', '59', '57',
'57',
'61', '59', '61', '61', '66']
```

Extraímos a temperatura máxima de cada dia e as armazenamos de modo organizado em uma lista na forma de strings.

Em seguida, converta essas strings em números usando `int()` para que

eles possam ser lidos pelo matplotlib:

highs_lows.py

```
  --trecho omitido--
      highs = []
      for row in reader:
u         high = int(row[1])
          highs.append(high)

      print(highs)
```

Convertemos as strings em inteiros em u antes de concatenar as temperaturas na lista. O resultado é uma lista das máximas diárias em formato numérico:

```
[64, 71, 64, 59, 69, 62, 61, 55, 57, 61, 57, 59, 57, 61, 64, 61, 59,
63, 60, 57,
69, 63, 62, 59, 57, 57, 61, 59, 61, 61, 66]
```

Agora vamos criar uma visualização para esses dados.

Plotando dados em um gráfico de temperatura

Para visualizar os dados de temperatura que temos, em primeiro lugar, vamos criar um gráfico simples das temperaturas máximas diárias usando o matplotlib, como vemos a seguir:

highs_lows.py

```
  import csv

  from matplotlib import pyplot as plt

  # Obtém as temperaturas máximas do arquivo
  --trecho omitido--

  # Faz a plotagem dos dados
  fig = plt.figure(dpi=128, figsize=(10, 6))
u plt.plot(highs, c='red')

  # Formata o gráfico
```

```
v plt.title("Daily high temperatures, July 2014", fontsize=24)
w plt.xlabel('', fontsize=16)
  plt.ylabel("Temperature (F)", fontsize=16)
  plt.tick_params(axis='both', which='major', labelsize=16)

  plt.show()
```

Passamos a lista de temperaturas máximas para `plot()` u e `c='red'` para plotar os pontos em vermelho. (Plotaremos as temperaturas máximas em vermelho e as mínimas em azul.) Então especificamos outros detalhes de formatação, por exemplo, o tamanho da fonte e os rótulos v, que você deverá reconhecer do Capítulo 15. Como ainda precisamos acrescentar as datas, não daremos um nome ao eixo x, mas `plt.xlabel()` modifica o tamanho da fonte para deixar os rótulos-padrão mais legíveis w. A Figura 16.1 mostra o gráfico resultante: um gráfico linear simples das temperaturas máximas em julho de 2014 para Sitka no Alasca.

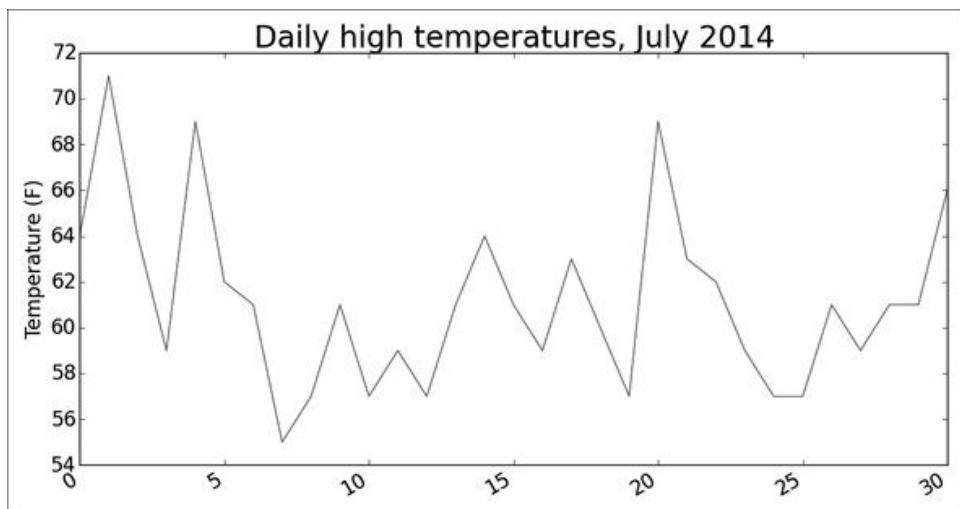



Figura 16.1 – Um gráfico linear que mostra as temperaturas máximas diárias em julho de 2014 para Sitka, no Alasca.

Módulo datetime

Vamos adicionar as datas em nosso gráfico para deixá-lo mais útil. A primeira data do arquivo de dados meteorológicos está na segunda linha do arquivo:

```
2014-7-1,64,56,50,53,51,48,96,83,58,30.19,--trecho omitido--
```

Os dados serão lidos na forma de string, portanto precisamos de um modo de converter a string '2014-7-1' em um objeto que represente essa data. Podemos construir um objeto que represente o dia 1 de julho de 2014 usando o método strptime() do módulo datetime. Vamos ver como strptime() funciona em uma sessão de terminal:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2014-7-1', '%Y-%m-%d')
>>> print(first_date)
2014-07-01 00:00:00
```

Inicialmente importamos a classe datetime do módulo datetime. Então chamamos o método strptime() com a string que contém a data com a qual queremos trabalhar, passando-a como o primeiro argumento. O segundo argumento informa a Python como a data está formatada. Nesse exemplo, '%Y-' diz a Python para interpretar a parte da string antes do primeiro traço como um ano contendo quatro dígitos; '%m-' diz a Python para interpretar a parte da string antes do segundo traço como um número que representa o mês; '%d' diz a Python para interpretar a última parte da string como o dia do mês, de 1 a 31.

O método strptime() aceita uma variedade de argumentos para determinar como a data deve ser interpretada. A Tabela 16.1 mostra alguns desses argumentos.

Tabela 16.1 – Argumentos para formatação de data e hora do módulo datetime

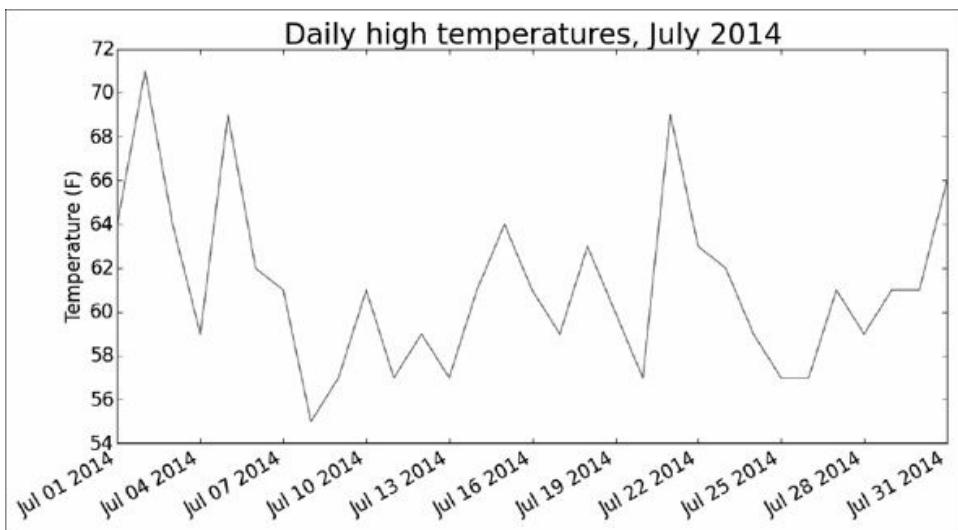
Argumento	Significado
%A	Nome do dia da semana, por exemplo, *Monday*
%B	Nome do mês, por exemplo, *January*
%m	Mês, como um número (de 01 a 12)
%d	Dia do mês, como um número (de 01 a 31)
%Y	Ano com quatro dígitos, por exemplo, 2015
%y	Ano com dois dígitos, por exemplo, 15
%H	Hora, no formato de 24 horas (de 00 a 23)
%I	Hora, no formato de 12 horas (de 01 a 12)
%p	am ou pm

%M	Minutos (de 00 a 59)
%S	Segundos (de 00 a 61)

Plotando datas

Agora que sabemos como processar as datas de nosso arquivo CSV, podemos melhorar nossa plotagem dos dados de temperatura extraindo as datas das máximas diárias e passando as datas e as temperaturas máximas para `plot()`, como vemos a seguir:

highs_lows.py

```
  import csv
  from datetime import datetime

  from matplotlib import pyplot as plt

  # Obtém as datas e as temperaturas máximas do arquivo
  filename = 'sitka_weather_07-2014.csv'
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

u     dates, highs = [], []
      for row in reader:
v         current_date = datetime.strptime(row[0], "%Y-%m-%d")
          dates.append(current_date)

          high = int(row[1])
          highs.append(high)

  # Faz a plotagem dos dados
  fig = plt.figure(dpi=128, figsize=(10, 6))
w plt.plot(dates, highs, c='red')

  # Formata o gráfico
  plt.title("Daily high temperatures, July 2014", fontsize=24)
  plt.xlabel('', fontsize=16)
x fig.autofmt_xdate()
  plt.ylabel("Temperature (F)", fontsize=16)
```

```
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

Criamos duas listas vazias para armazenar as datas e as temperaturas máximas do arquivo **u**. Então convertemos os dados que contêm as informações de datas (`row[0]`) em um objeto `datetime` **v** e o concatenamos em `dates`. Passamos as datas e os valores das temperaturas máximas para `plot()` em **w**. A chamada a `fig.autofmt_xdate()` em **x** desenha os rótulos com as datas na diagonal para evitar que se sobreponham. A Figura 16.2 mostra o gráfico melhorado.

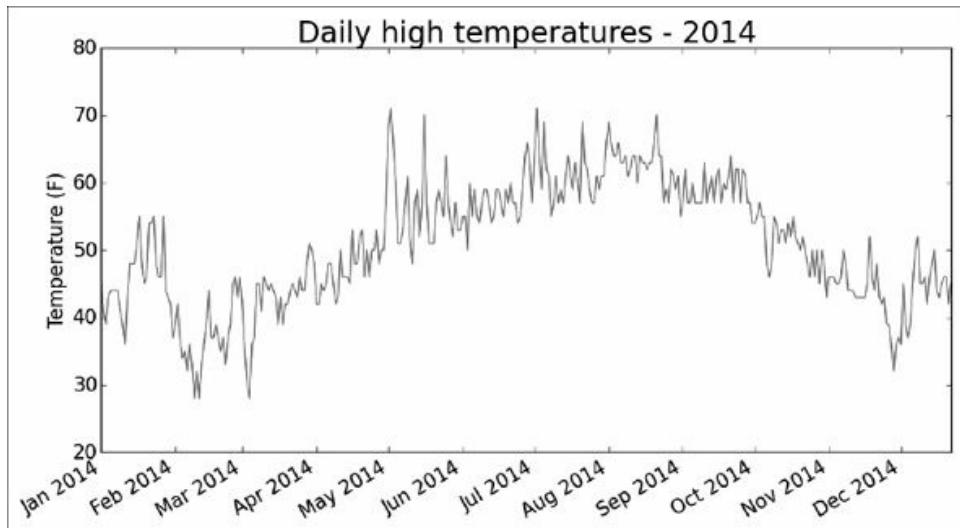
Figura 16.2 – O gráfico está mais significativo, agora que mostra as datas no eixo x.

Plotando um período de tempo maior

Com o nosso gráfico configurado, vamos acrescentar mais dados para obter uma imagem mais completa do clima em Sitka. Copie o arquivo *sitka_weather_2014.csv*, que contém dados do Weather Underground para Sitka durante um ano, para a pasta em que você está armazenando os programas deste capítulo.

Agora podemos gerar um gráfico para o clima do ano todo:

highs_lows.py

```
--trecho omitido--
# Obtém as datas e as temperaturas máximas do arquivo
```

```
u filename = 'sitka_weather_2014.csv'
  with open(filename) as f:
  --trecho omitido--
  # Formata o gráfico
v plt.title("Daily high temperatures - 2014", fontsize=24)
  plt.xlabel('', fontsize=16)
  --trecho omitido--
```

Modificamos o nome do arquivo para usar o novo arquivo de dados *sitka_weather_2014.csv* u e atualizamos o título de nosso gráfico para que reflita a mudança em seu conteúdo v. A Figura 16.3 mostra o gráfico resultante.

Figura 16.3 – Dados de um ano.

Plotando uma segunda série de dados

O gráfico revisado da Figura 16.3 mostra uma quantidade substancial de dados significativos, mas podemos deixá-lo mais útil ainda se incluirmos as temperaturas mínimas. Precisamos extrair as temperaturas mínimas do arquivo de dados e então adicioná-las ao nosso gráfico, como vemos a seguir:

highs_lows.py

```
  --trecho omitido--
  # Obtém as datas e as temperaturas máximas e mínimas do arquivo
  filename = 'sitka_weather_2014.csv'
```

```
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

u     dates, highs, lows = [], [], []
      for row in reader:
          current_date = datetime.strptime(row[0], "%Y-%m-%d")
          dates.append(current_date)

          high = int(row[1])
          highs.append(high)

v         low = int(row[3])
          lows.append(low)

  # Faz a plotagem dos dados
  fig = plt.figure(dpi=128, figsize=(10, 6))
  plt.plot(dates, highs, c='red')
w plt.plot(dates, lows, c='blue')

  # Formata o gráfico
x plt.title("Daily high and low temperatures - 2014", fontsize=24)
  --trecho omitido--
```

Em u adicionamos a lista vazia `lows` para armazenar as temperaturas mínimas e então extraímos e armazenamos essas temperaturas para cada data, a partir da quarta posição de cada linha (`row[3]`) v. Em w acrescentamos uma chamada a `plot()` para as temperaturas mínimas e colorimos esses valores de azul. Por fim, atualizamos o título x. A Figura 16.4 mostra o gráfico resultante.

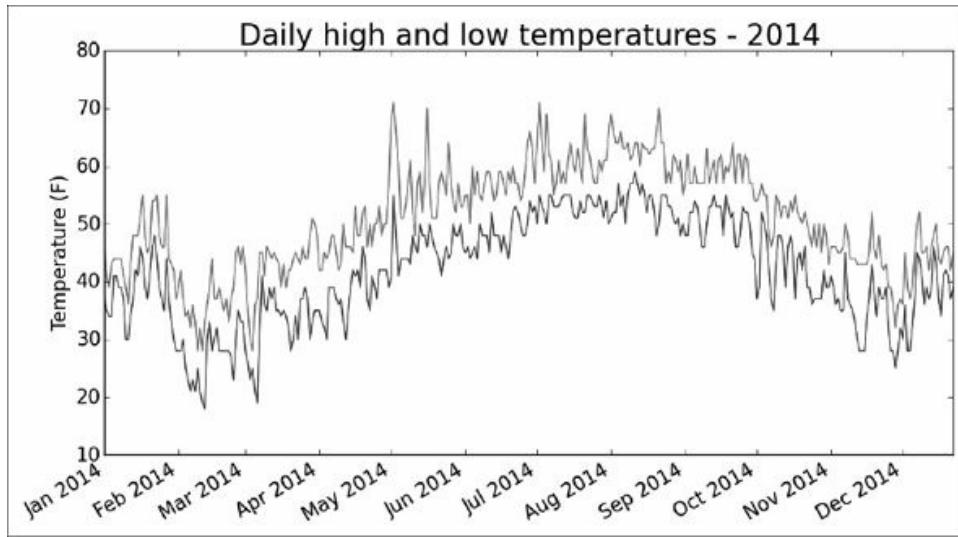



Figura 16.4 – Duas séries de dados no mesmo gráfico.

Sombreando uma área do gráfico

Depois de ter adicionado duas séries de dados, podemos agora analisar a variação de temperatura para cada dia. Vamos acrescentar um toque final no gráfico usando sombreamento para mostrar a variação entre as temperaturas mínima e máxima a cada dia. Para isso usaremos o método `fill_between()`, que aceita uma série de valores de x e duas séries de valores de y, e preenche o espaço entre as duas séries de valores de y:

highs_lows.py

```
  --trecho omitido--
  # Faz a plotagem dos dados
  fig = plt.figure(dpi=128, figsize=(10, 6))
u plt.plot(dates, highs, c='red', alpha=0.5)
  plt.plot(dates, lows, c='blue', alpha=0.5)
v plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
  --trecho omitido--
```

O argumento `alpha` em u controla a transparência de uma cor. Um valor de `alpha` igual a 0 é totalmente transparente e 1 (o default) é totalmente opaco. Ao definir `alpha` com 0,5, fazemos as linhas vermelha e azul do gráfico serem mais claras.

Em v passamos para `fill_between()` a lista `dates` para os valores de x e então as duas séries com valores de y, `highs` e `lows`. O argumento `facecolor`

determina a cor da região sombreada, e lhe fornecemos um valor baixo de `alpha`, igual a 0,1, para que a região preenchida conecte as duas séries de dados sem provocar distrações na informação que elas representam. A Figura 16.5 mostra o gráfico com a região sombreada entre as temperaturas máximas e mínimas.

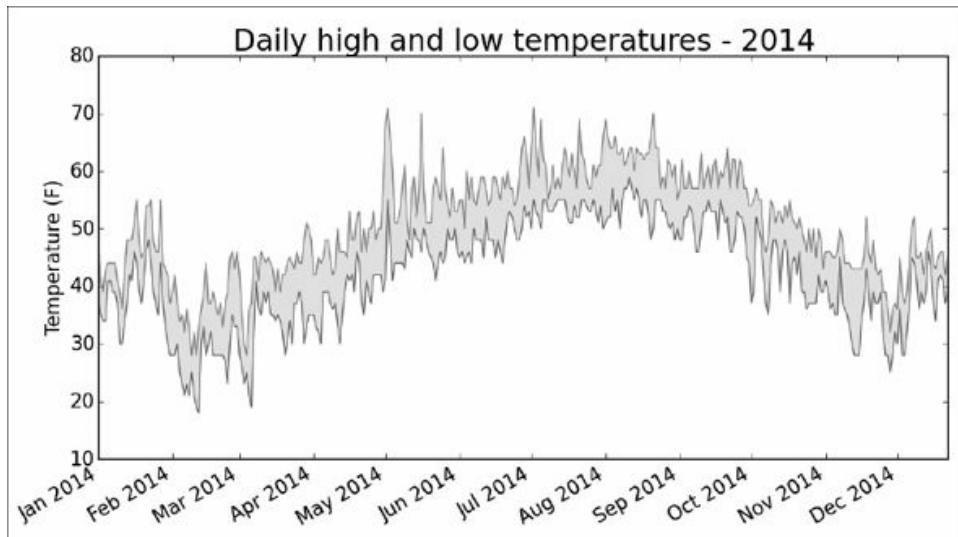



Figura 16.5 – A região entre os dois conjuntos de dados está sombreada.

O sombreamento ajuda a deixar o intervalo entre os dois conjuntos de dados imediatamente aparente.

Verificação de erros

Devemos ser capazes de executar o código de *highs_lows.py* usando dados de qualquer localidade. Porém, algumas estações meteorológicas ocasionalmente funcionam mal e falham em coletar alguns dos dados que deveriam obter – ou todos eles. A ausência de dados pode resultar em exceções capazes de causar falhas em nossos programas se não as tratarmos de forma apropriada.

Por exemplo, vamos ver o que acontece quando tentamos gerar um gráfico de temperaturas para o Vale da Morte na Califórnia. Copie o arquivo *death_valley_2014.csv* para a pasta em que você está armazenando os programas deste capítulo e altere *highs_lows.py* para que gere um gráfico para o Vale da Morte:

highs_lows.py

```
--trecho omitido--
# Obtém as datas e as temperaturas máximas e mínimas do arquivo
filename = 'death_valley_2014.csv'
with open(filename) as f:
--trecho omitido--
```

Quando executamos o programa, vemos um erro, conforme mostrado na última linha da saída a seguir:

```
Traceback (most recent call last):
  File "highs_lows.py", line 17, in <module>
    high = int(row[1])
ValueError: invalid literal for int() with base 10: ''
```

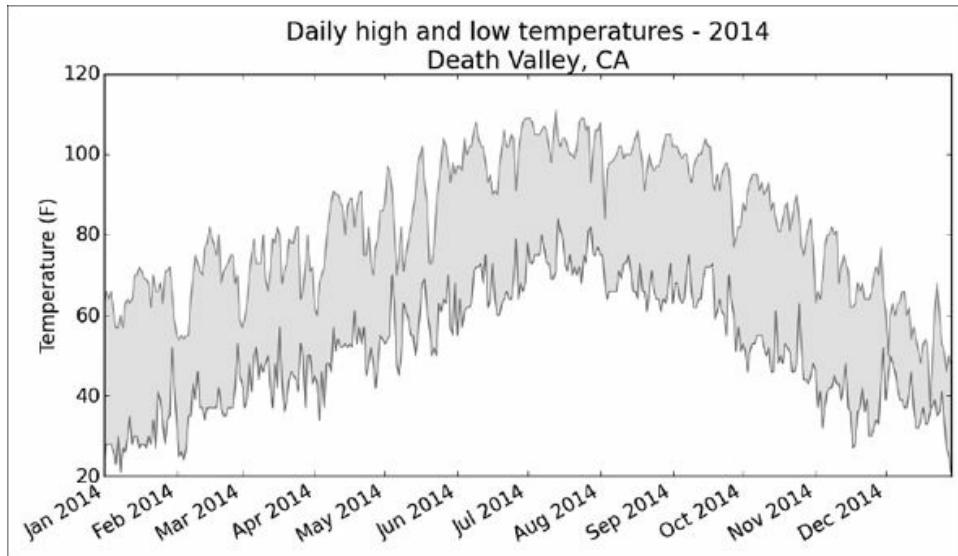
O traceback nos informa que Python não foi capaz de processar a temperatura máxima para uma das datas porque não é possível transformar uma string vazia ('') em um inteiro. Uma observação em *death_valley_2014.csv* mostra qual é o problema:

```
2014-2-16,,,,,,,,,,,,,,,,,,,0.00,,,-1
```

Parece que em 16 de fevereiro de 2014, nenhum dado foi registrado; a string para a temperatura máxima está vazia. Para cuidar desse problema, executaremos um código de verificação de erros quando os valores forem lidos do arquivo CSV para tratar exceções que possam surgir no parse de nossos conjuntos de dados. Eis o modo como isso funciona:

highs_lows.py

```
  --trecho omitido--
  # Obtém as datas e as temperaturas máximas e mínimas do arquivo
  filename = 'death_valley_2014.csv'
  with open(filename) as f:
      reader = csv.reader(f)
      header_row = next(reader)

      dates, highs, lows = [], [], []
      for row in reader:
u         try:
              current_date = datetime.strptime(row[0], "%Y-%m-%d")
              high = int(row[1])
```

```
            low = int(row[3])
        except ValueError:
v           print(current_date, 'missing data')
        else:
w           dates.append(current_date)
            highs.append(high)
            lows.append(low)

# Faz a plotagem dos dados
--trecho omitido--

# Formata o gráfico
x title = "Daily high and low temperatures - 2014\nDeath Valley, CA"
plt.title(title, fontsize=20)
--trecho omitido--
```

Sempre que analisamos uma linha, tentamos extrair a data e as temperaturas máxima e mínima u. Se houver algum dado faltando, Python levantará um `ValueError` e o trataremos exibindo uma mensagem de erro que inclua a data do dado ausente v. Depois de exibir o erro, o laço continuará processando a próxima linha. Se todos os dados para uma data forem recuperados sem erro, o bloco `else` será executado e os dados serão concatenados nas listas apropriadas w. Como estamos plotando informações para uma nova localidade, atualizamos o título para que o nome do lugar seja incluído no gráfico x.

Quando executar *highs_lows.py* agora, você verá que estão faltando dados apenas para uma data:

```
2014-02-16 missing data
```

A Figura 16.6 mostra o gráfico resultante.

Figura 16.6 – Temperaturas máximas e mínimas diárias para o Vale da Morte.

Se compararmos esse gráfico com o gráfico de Sitka, podemos ver que o Vale da Morte é, de modo geral, mais quente que o sudeste do Alasca, como esperado, mas a variação da temperatura a cada dia, na verdade, é maior no deserto. A altura da região sombreada deixa isso claro.

Muitos conjuntos de dados com os quais você trabalhar terão dados faltando, dados formatados de maneira inapropriada ou dados incorretos. Utilize as ferramentas que você conheceu na primeira metade deste livro para lidar com essas situações. Nesse caso usamos um bloco `try-except-else` para tratar os dados ausentes. Às vezes você usará `continue` para ignorar alguns dados ou utilizará `remove()` ou `del` para eliminá-los depois que forem extraídos. Qualquer abordagem que funcione pode ser usada, desde que o resultado seja uma visualização precisa e significativa.

FAÇA VOCÊ MESMO

16.1 – São Francisco: As temperaturas em São Francisco são mais parecidas com as temperaturas em Sitka ou com as temperaturas no Vale da Morte? Gere um gráfico com as temperaturas máximas e mínimas de São Francisco e faça uma comparação. (Você pode fazer download de dados meteorológicos de praticamente qualquer localidade a partir de *http://www.wunderground.com/history/*. Especifique uma localidade e o intervalo de datas, faça rolagens para o final da página e localize um link chamado *Comma-Delimited File* (Arquivo delimitado por vírgula). Clique com o botão direito

do mouse nesse link e salve os dados em um arquivo CSV.)

16.2 – Comparação entre Sitka e o Vale da Morte: As escalas de temperatura nos gráficos de Sitka e do Vale da Morte refletem os diferentes intervalos de dados. Para comparar as variações de temperatura entre Sitka e o Vale da Morte de modo preciso, é necessário usar escalas idênticas no eixo y. Mude as configurações do eixo y em um dos gráficos das Figuras 16.5 e 16.6, ou em ambos, e faça uma comparação direta entre as variações de temperatura em Sitka e no Vale da Morte (ou de quaisquer dois lugares que você queira comparar). Você também poderá tentar plotar os dois conjuntos de dados no mesmo gráfico.

16.3 – Índice pluviométrico: Escolha qualquer lugar que seja de seu interesse e crie uma visualização que apresente o índice pluviométrico em um gráfico. Comece se concentrando nos dados de um mês e, em seguida, depois que seu código estiver funcionando, execute-o para os dados de um ano todo.

16.4 – Explore: Gere mais algumas visualizações que analisem qualquer outro aspecto climático de seu interesse para qualquer localidade sobre a qual você tenha curiosidade.

Mapeando conjuntos de dados globais: formato JSON

Nesta seção você fará o download de dados de diferentes países em formato JSON e trabalhará com esses dados usando o módulo `json`. Utilizando a ferramenta de mapeamento amigável a iniciantes do Pygal para dados baseados em países, você criará visualizações desses dados a fim de explorar padrões globais que dizem respeito à distribuição da população mundial por diferentes países.

Fazendo download dos dados da população mundial

Copie o arquivo *population_data.json*, que contém dados sobre as populações da maior parte dos países do mundo, de 1960 a 2010, para a pasta em que você está armazenando os programas deste capítulo. Esses dados são provenientes de um dos muitos conjuntos de dados que a Open Knowledge Foundation (*http://data.okfn.org/*) disponibiliza gratuitamente.

Extraindo dados relevantes

Vamos observar o arquivo *population_data.json* para ver como podemos começar a processar seus dados:

population_data.json

```
[
  {
    "Country Name": "Arab World",
    "Country Code": "ARB",
    "Year": "1960",
    "Value": "96388069"
  },
  {
    "Country Name": "Arab World",
    "Country Code": "ARB",
    "Year": "1961",
    "Value": "98882541.4"
  },
  --trecho omitido--
]
```

O arquivo é composto basicamente de uma longa lista Python. Cada item é um dicionário com quatro chaves: o nome de um país, o seu código, um ano e um valor que representa a população. Queremos analisar o nome de cada país e a população somente em 2010, portanto começaremos escrevendo um programa que exiba apenas essas informações:

world_population.py

```
  import json

  # Carrega os dados em uma lista
  filename = 'population_data.json'
  with open(filename) as f:
u     pop_data = json.load(f)

  # Exibe a população de cada país em 2010
v for pop_dict in pop_data:
w     if pop_dict['Year'] == '2010':
x         country_name = pop_dict['Country Name']
          population = pop_dict['Value']
          print(country_name + ": " + population)
```

Inicialmente importamos o módulo `json` para que seja possível carregar

os dados do arquivo de forma apropriada e, em seguida, armazenamos esses dados em `pop_data` em `u`. A função `json.load()` converte os dados em um formato com que Python possa trabalhar: nesse caso, em uma lista. Em `v` percorremos cada item de `pop_data` com um laço. Cada item é um dicionário com quatro pares chave-valor, e o armazenamos em `pop_dict`.

Em `w` procuramos 2010 na chave `'Year'` de cada dicionário. (Como os valores em *population_data.json* estão todos entre aspas, fazemos uma comparação entre strings.) Se o ano for 2010, armazenamos o valor associado a `'Country Name'` em `country_name` e o valor associado a `'Value'` em `population` em `x`. Então exibimos o nome de cada país e a sua população.

A saída é composta de uma série de nomes de países e a população:

```
Arab World: 357868000
Caribbean small states: 6880000
East Asia & Pacific (all income levels): 2201536674
--trecho omitido--
Zimbabwe: 12571000
```

Nem todos os dados capturados incluem os nomes exatos dos países, mas esse é um bom ponto de partida. Agora precisamos converter os dados em um formato com que o Pygal possa trabalhar.

Convertendo strings em valores numéricos

Todas as chaves e valores em *population_data.json* estão armazenados como strings. Para trabalhar com os dados referentes às populações, devemos converter as strings com as populações em valores numéricos. Fazemos isso usando a função `int()`:

world_population.py

```
--trecho omitido--
for pop_dict in pop_data:
    if pop_dict['Year'] == '2010':
        country_name = pop_dict['Country Name']
u         population = int(pop_dict['Value'])
v         print(country_name + ": " + str(population))
```

Agora armazenamos o valor de cada população em um formato numérico em u. Quando exibimos o valor da população, devemos convertê-lo para uma string em v.

No entanto, essa mudança resulta em um erro para alguns valores, como vemos a seguir:

```
Arab World: 357868000
Caribbean small states: 6880000
East Asia & Pacific (all income levels): 2201536674
--trecho omitido--
Traceback (most recent call last):
  File "print_populations.py", line 12, in <module>
    population = int(pop_dict['Value'])
u ValueError: invalid literal for int() with base 10: '1127437398.85751'
```

Com frequência, dados brutos não estão formatados de forma consistente, portanto podemos nos deparar com vários erros. Nesse caso, o erro ocorreu porque Python não é capaz de transformar uma string contendo um decimal, `'1127437398.85751'`, diretamente em um inteiro u. (Esse valor decimal provavelmente é o resultado da interpolação para os anos em que a contagem de uma população específica não foi feita.) Tratamos esse erro convertendo a string em um número de ponto flutuante e então convertendo esse número em um inteiro:

world_population.py

```
--trecho omitido--
for pop_dict in pop_data:
    if pop_dict['Year'] == '2010':
        country = pop_dict['Country Name']
        population = int(float(pop_dict['Value']))
        print(country + ": " + str(population))
```

A função `float()` transforma a string em um decimal, e a função `int()` remove a parte decimal do número e devolve um inteiro. Agora podemos exibir um conjunto completo de valores de populações para o ano de 2010, sem erros:

```
Arab World: 357868000
Caribbean small states: 6880000
```

```
East Asia & Pacific (all income levels): 2201536674
--trecho omitido--
Zimbabwe: 12571000
```

Cada string foi convertida com sucesso em um número de ponto flutuante e depois para um inteiro. Agora que os valores dessas populações estão armazenados em um formato numérico, podemos usá-los para criar um mapa da população mundial.

Obtendo os códigos de duas letras dos países

Antes de podermos nos concentrar no mapeamento, devemos tratar um último aspecto dos dados. A ferramenta de mapeamento do Pygal espera dados em um formato particular: os países devem ser fornecidos na forma de códigos e as populações como valores. Vários conjuntos padronizados de códigos de países são usados com frequência quando trabalhamos com dados geopolíticos; os códigos incluídos em *population_data.json* são códigos de três letras, mas o Pygal utiliza códigos de duas letras. Precisamos de uma maneira de descobrir o código de duas letras a partir do nome do país.

Os códigos dos países no Pygal estão armazenados em um módulo chamado `i18n`, que é uma abreviatura para *internationalization* (internacionalização). O dicionário `COUNTRIES` contém os códigos de duas letras dos países como chaves e os nomes dos países como valores. Para ver esses códigos, importe o dicionário do módulo `i18n` e exiba suas chaves e valores:

countries.py

```
  from pygal.i18n import COUNTRIES

u for country_code in sorted(COUNTRIES.keys()):
      print(country_code, COUNTRIES[country_code])
```

No laço `for`, dizemos a Python para colocar as chaves em ordem alfabética u.

Então exibimos o código de cada país e o nome associado a ele:

```
ad Andorra
```

```
ae United Arab Emirates
af Afghanistan
--trecho omitido--
zw Zimbabwe
```

Para extrair o código do país, escrevemos uma função que faz uma pesquisa em `COUNTRIES` e devolve esse código. Escreveremos esse código em um módulo separado chamado `country_codes` para que possamos importá-lo depois em um programa de visualização:

country_codes.py

```
  from pygal.i18n import COUNTRIES

u def get_country_code(country_name):
      """Devolve o código de duas letras do Pygal para um país, dado o
  seu nome."""
v     for code, name in COUNTRIES.items():
w         if name == country_name:
              return code
      # Se o país não foi encontrado, devolve None
x     return None

  print(get_country_code('Andorra'))
  print(get_country_code('United Arab Emirates'))
  print(get_country_code('Afghanistan'))
```

Passamos o nome do país para `get_country_code()` e o armazenamos no parâmetro `country_name` u. Em seguida, percorremos os pares de código-nome do país em `COUNTRIES` com um laço v. Se o nome do país for encontrado, o código desse país será devolvido w. Adicionamos uma linha depois do laço para devolver `None` se o nome do país não for encontrado x. Por fim, passamos os nomes de três países para conferir se a função está correta. Como esperado, o programa mostra os códigos de duas letras para três países:

```
ad
ae
af
```

Antes de usar essa função, remova as instruções `print` de

country_codes.py.

Em seguida, importamos `get_country_code()` em *world_population.py*:

world_population.py

```
  import json

  from country_codes import get_country_code
  --trecho omitido--

  # Exibe a população de cada país em 2010
  for pop_dict in pop_data:
      if pop_dict['Year'] == '2010':
          country_name = pop_dict['Country Name']
          population = int(float(pop_dict['Value']))
u         code = get_country_code(country_name)
          if code:
v             print(code + ": "+ str(population))
w         else:
              print('ERROR - ' + country_name)
```

Depois de extrair o nome do país e a população, armazenamos o código desse país em `code`, ou `None` se nenhum código estiver disponível u. Se um código for devolvido, o código e a população do país serão exibidos v. Se o código não estiver disponível, exibimos uma mensagem de erro com o nome do país cujo código não encontramos w. Execute o programa e você verá alguns códigos de países com suas populações e algumas linhas com erro:

```
ERROR - Arab World
ERROR - Caribbean small states
ERROR - East Asia & Pacific (all income levels)
--trecho omitido--
af: 34385000
al: 3205000
dz: 35468000
--trecho omitido--
ERROR - Yemen, Rep.
zm: 12927000
zw: 12571000
```

Os erros têm duas origens. Em primeiro lugar, nem todas as classificações no conjunto de dados são feitas por país; algumas estatísticas de população são para regiões (*Arab World*, ou Mundo Árabe) e grupos econômicos (*all income levels*, ou todos os níveis de renda). Em segundo lugar, algumas das estatísticas usam um sistema diferente para nomes completos de países (*Yemen, Rep.* em vez de *Yemen*). Por enquanto, omitiremos os dados de países que causam erros e veremos como será a aparência de nosso mapa para os dados recuperados com sucesso.

Construindo um mapa-múndi

Com os códigos dos países que temos, criar um mapa-múndi é rápido e fácil. O Pygal inclui um tipo de mapa chamado `Worldmap` para ajudar a mapear conjuntos de dados globais. Como exemplo de como usar o `Worldmap`, criaremos um mapa simples que destaque a América do Norte, a América Central e a América do Sul:

americas.py

```
  import pygal

u wm = pygal.Worldmap()
  wm.title = 'North, Central, and South America'

v wm.add('North America', ['ca', 'mx', 'us'])
  wm.add('Central America', ['bz', 'cr', 'gt', 'hn', 'ni', 'pa', 'sv'])
  wm.add('South America', ['ar', 'bo', 'br', 'cl', 'co', 'ec', 'gf',
      'gy', 'pe', 'py', 'sr', 'uy', 've'])

w wm.render_to_file('americas.svg')
```

Em u criamos uma instância da classe `Worldmap` e definimos o atributo `title` do mapa. Em v usamos o método `add()`, que aceita um rótulo e uma lista de códigos de países que queremos destacar. Cada chamada a `add()` define uma nova cor para o conjunto de países e acrescenta essa cor a uma legenda à esquerda da imagem, com o rótulo especificado nessa chamada. Queremos que toda a região da América do Norte seja representada com uma cor, portanto colocamos `'ca'`, `'mx'` e `'us'` na lista que passamos para a

primeira chamada a `add()` a fim de dar destaque ao Canadá, ao México e aos Estados Unidos em conjunto. Então fizemos o mesmo para os países da América Central e da América do Sul.

O método `render_to_file()` em `W` cria um arquivo *.svg* contendo o mapa, que poderá ser aberto em seu navegador. A saída é um mapa que destaca a América do Norte, a América Central e a América do Sul com cores diferentes, como mostra a Figura 16.7.

Figura 16.7 – Uma instância simples do tipo `Worldmap`.

Agora sabemos como criar um mapa com áreas coloridas, uma legenda e rótulos claros. Vamos adicionar dados ao nosso mapa para mostrar informações sobre um país.

Plotando dados numéricos em um mapa-múndi

Para exercitar a forma de colocar dados numéricos em um mapa, crie um mapa que mostre as populações dos três países da América do Norte:

na_populations.py

```
  import pygal

  wm = pygal.Worldmap()
  wm.title = 'Populations of Countries in North America'
u wm.add('North America', {'ca': 34126000, 'us': 309349000, 'mx':
113423000})

  wm.render_to_file('na_populations.svg')
```

Inicialmente crie uma instância de `Worldmap` e defina um título. Então utilize o método `add()`, mas desta vez passe um dicionário como segundo argumento em vez de passar uma lista u. O dicionário contém os códigos de duas letras do Pygal para os países como chaves e as populações como valores. O Pygal utiliza automaticamente esses números para sombrear os países, variando da cor mais clara (menos populosos) para a cor mais escura (mais populosos). A Figura 16.8 mostra o mapa resultante.

Figura 16.8 – População dos países da América do Norte.

Esse mapa é interativo: se passar o mouse sobre cada país, você verá a sua população. Vamos acrescentar mais dados em nosso mapa.

Criando um mapa completo de populações

Para plotar os valores das populações para o restante dos países, devemos converter os dados dos países que processamos anteriormente em um formato de dicionário esperado pelo Pygal, que é composto dos códigos de duas letras para os países como chaves e as populações como valores. Acrescente o código a seguir em *world_population.py*:

world_population.py

```
  import json

  import pygal

  from country_codes import get_country_code

  # Carrega os dados em uma lista
  --trecho omitido--

  # Constrói um dicionário com dados das populações
u cc_populations = {}
  for pop_dict in pop_data:
      if pop_dict['Year'] == '2010':
          country = pop_dict['Country Name']
          population = int(float(pop_dict['Value']))
          code = get_country_code(country)
          if code:
v             cc_populations[code] = population

w wm = pygal.Worldmap()
  wm.title = 'World Population in 2010, by Country'
x wm.add('2010', cc_populations)

  wm.render_to_file('world_population.svg')
```

Inicialmente importe o `pygal`. Em u criamos um dicionário vazio para armazenar os códigos dos países e as populações no formato esperado pelo Pygal. Em v construímos o dicionário `cc_populations` usando o código do país como chave e a população como valor sempre que um código é devolvido. Também removemos todas as instruções `print`.

Criamos uma instância de `Worldmap` e definimos seu atributo `title` W. Quando chamamos `add()`, passamos o dicionário com os códigos dos países e as populações para esse método X. A Figura 16.9 mostra o mapa gerado.

Figura 16.9 – A população mundial em 2010.

Não temos dados para alguns países, que estão em preto, mas a maioria deles está colorida de acordo com o tamanho de sua população. Você lidará com dados ausentes mais adiante neste capítulo, mas antes disso vamos alterar o sombreamento para que reflita de modo mais exato a população dos países. No momento, nosso mapa mostra muitos países com tons claros e dois países com tons bem escuros. O contraste entre a maioria dos países não é suficiente para mostrar quão populosos eles são, uns em relação aos outros. Corrigiremos isso agrupando os países em níveis de população e sombreando cada grupo.

Agrupando os países de acordo com a sua população

Como a China e a Índia são bem mais populosas que os outros países, o

mapa mostra pouco contraste. A China e a Índia têm, cada uma delas, mais de um bilhão de pessoas, enquanto o próximo país mais populoso são os Estados Unidos, com aproximadamente 300 milhões de pessoas. Em vez de plotar todos os países como um grupo, vamos separá-los em três níveis populacionais: menos de 10 milhões, entre 10 milhões e 1 bilhão e acima de 1 bilhão.

world_population.py

```
  --trecho omitido--
  # Constrói um dicionário com dados das populações
  cc_populations = {}
  for pop_dict in pop_data:
      if pop_dict['Year'] == '2010':
          --trecho omitido--
          if code:
              cc_populations[code] = population

  # Agrupa os países em três níveis populacionais
u cc_pops_1, cc_pops_2, cc_pops_3 = {}, {}, {}
v for cc, pop in cc_populations.items():
      if pop < 10000000:
          cc_pops_1[cc] = pop
      elif pop < 1000000000:
          cc_pops_2[cc] = pop
      else:
          cc_pops_3[cc] = pop

  # Vê quantos países estão em cada nível
w print(len(cc_pops_1), len(cc_pops_2), len(cc_pops_3))

  wm = pygal.Worldmap()
  wm.title = 'World Population in 2010, by Country'
x wm.add('0-10m', cc_pops_1)
  wm.add('10m-1bn', cc_pops_2)
  wm.add('>1bn', cc_pops_3)

  wm.render_to_file('world_population.svg')
```

Para agrupar os países, criamos um dicionário vazio para cada categoria

u. Então percorremos `cc_populations` com um laço a fim de verificar a população de cada país v. O bloco `if-elif-else` adiciona uma entrada no dicionário apropriado (`cc_pops_1`, `cc_pops_2` ou `cc_pops_3`) para cada par código de país-população.

Em w exibimos o tamanho de cada um desses dicionários para descobrir o tamanho dos grupos. Quando plotamos os dados x, garantimos que todos os três grupos sejam adicionados em `Worldmap`. Ao executar esse programa, você verá inicialmente o tamanho de cada grupo:

```
85 69 2
```

Essa saída mostra que há 85 países com menos de 10 milhões de pessoas, 69 países com população entre 10 milhões e 1 bilhão de pessoas e dois países extremos com mais de 1 bilhão. Parece ser uma divisão uniforme o suficiente para um mapa informativo. A Figura 16.10 mostra o mapa resultante.

Figura 16.10 – A população mundial exibida em três grupos diferentes.

Agora três cores diferentes nos ajudam a ver as distinções entre os níveis

populacionais. Em cada um desses três níveis, os países são sombreados da cor clara para a cor mais escura, correspondendo à variação das populações menores para as maiores.

Estilizando mapas-múndi com o Pygal

Embora os grupos populacionais em nosso mapa sejam eficientes, as configurações default para as cores não são bonitas: por exemplo, nesse caso, o Pygal escolheu um rosa-choque e um tom de verde. Usaremos as diretivas de estilização do Pygal para corrigir as cores.

Vamos orientar o Pygal para que use uma cor de base novamente, porém desta vez escolheremos a cor e aplicaremos um sombreamento mais distinto para os três grupos populacionais:

world_population.py

```
  import json

  import pygal
u from pygal.style import RotateStyle
  --trecho omitido--
  # Agrupa os países em três níveis populacionais
  cc_pops_1, cc_pops_2, cc_pops_3 = {}, {}, {}
  for cc, pop in cc_populations.items():
      if pop < 10000000:
          --trecho omitido--

v wm_style = RotateStyle('#336699')
w wm = pygal.Worldmap(style=wm_style)
  wm.title = 'World Population in 2010, by Country'
  --trecho omitido--
```

Os estilos do Pygal estão armazenados no módulo `style` do qual importamos o estilo `RotateStyle` u. Essa classe aceita um argumento, que é uma cor RGB em formato hexa v. O Pygal então escolhe as cores para cada um dos grupos de acordo com a cor fornecida. O *formato hexa* é uma string com um sinal de sustenido (`#`), seguido de seis caracteres: os dois primeiros representam o componente vermelho da cor, os próximos dois representam o componente verde e os dois últimos representam o

componente azul. Os valores dos componentes podem variar de `00` (nada dessa cor) a `FF` (quantidade máxima dessa cor). Se pesquisar os termos *hex color chooser* (hexa cor selecionador) online, você deverá encontrar uma ferramenta que permita fazer experimentos com as cores e lhe forneça os valores RGB. A cor usada nesse caso (#336699) mistura um pouco de vermelho (33), um pouco mais de verde (66) e mais ainda de azul (99), o que resulta em uma cor azul clara como base para `RotateStyle` trabalhar.

`RotateStyle` devolve um objeto estilo, que armazenamos em `wm_style`. Para usar esse objeto, passe-o como um argumento nomeado ao criar uma instância de `Worldmap W`. A Figura 16.11 mostra o mapa atualizado.

Figura 16.11 – Os três grupos populacionais com um tema unificado pela cor.

Essa estilização proporciona uma aparência uniforme ao mapa e resulta em grupos que são fáceis de distinguir.

Clareando a cor do tema

O Pygal tende a usar temas escuros, por padrão. Para fazer a exibição,

deixei o estilo de meus mapas mais claros usando `LightColorizedStyle`. Essa classe altera o tema do mapa como um todo, incluindo a cor de fundo e os rótulos, assim como as cores individuais dos países. Para usá-la, inicialmente importe o estilo:

```
from pygal.style import LightColorizedStyle
```

Então você pode usar `LightColorizedStyle` sozinho, assim:

```
wm_style = LightColorizedStyle
```

No entanto, essa classe não oferece nenhum controle direto a você sobre a cor usada, portanto o Pygal escolherá uma cor base default. Para definir uma cor, utilize `LightColorizedStyle` como base para `RotateStyle`. Importe `LightColorizedStyle` e `RotateStyle`:

```
from pygal.style import LightColorizedStyle, RotateStyle
```

Em seguida crie um estilo usando `RotateStyle`, mas passe um argumento adicional `base_style`:

```
wm_style = RotateStyle('#336699', base_style=LightColorizedStyle)
```

Isso resulta em um tema, de modo geral, mais claro, porém as cores dos países serão baseadas na cor que você passar como argumento. Com esse estilo, você verá que seus mapas se parecerão mais com as imagens de tela vistas aqui.

Enquanto fizer experimentos para descobrir quais são as diretivas de estilo que funcionam melhor para diferentes visualizações, usar aliases em suas instruções `import` poderá ajudar:

```
from pygal.style import LightColorizedStyle as LCS, RotateStyle as RS
```

Isso resultará em definições de estilo mais concisas:

```
wm_style = RS('#336699', base_style=LCS)
```

Ao usar apenas esse pequeno conjunto de diretivas de estilização, você terá um controle significativo da aparência dos gráficos e dos mapas no Pygal.

FAÇA VOCÊ MESMO

16.5 – Todos os países: Nos mapas de população que criamos nesta seção, nosso programa não foi capaz de encontrar automaticamente os códigos de duas letras para aproximadamente 12 países. Descubra quais são os países com códigos ausentes e procure-os no dicionário `COUNTRIES`. Acrescente um bloco **`if-elif`** em **`get_country_code()`** para que ele devolva os valores corretos dos códigos desses países específicos:

```
if country_name == 'Yemen, Rep.'
    return 'ye'
elif --trecho omitido--
```

Coloque esse código depois do laço em `COUNTRIES`, mas antes da instrução `return None`. Quando terminar, você deverá ver um mapa mais completo.

16.6 – Produto Interno Bruto: A Open Knowledge Foundation mantém um conjunto de dados contendo o PIB – ou GDP (Gross Domestic Product) – de cada país, que pode ser encontrado em *http://data.okfn.org/data/core/gdp/*. Faça o download da versão JSON desse conjunto de dados e plote o PIB do último ano de cada país.

16.7 – Escolha os seus próprios dados: O Banco Mundial (World Bank) mantém vários conjuntos de dados separados com informações sobre cada país. Acesse *http://data.worldbank.org/indicator/* e encontre um conjunto de dados que pareça ser interessante. Clique no conjunto de dados, depois no link Download Data (Dados para download) e escolha CSV. Você receberá três arquivos CSV, dois dos quais contêm *Metadata* no nome; utilize o terceiro arquivo CSV. Escreva um programa que gere um dicionário com os códigos de duas letras do Pygal para os países como chaves e o dado que você escolheu no arquivo como valores. Plote os dados em um `Worldmap` e estilize-o como quiser.

16.8 – Testando o módulo **`country_codes`**: Quando escrevemos o módulo `country_codes`, utilizamos instruções `print` para verificar se a função `get_country_code()` estava correta. Escreva um teste apropriado para essa função usando o que você aprendeu no Capítulo 11.

Resumo

Neste capítulo aprendemos a trabalhar com conjuntos de dados online. Vimos como processar arquivos CSV e JSON, além de extrair os dados em que queríamos colocar o foco. Usando dados meteorológicos históricos, aprendemos mais sobre como trabalhar com o matplotlib, incluindo a

utilização do módulo `datetime` e o modo de plotar várias séries de dados em um único gráfico. Vimos como plotar dados de países em um mapa-múndi no Pygal e a estilizar mapas e gráficos desse módulo.

À medida que adquirir mais experiência com arquivos CSV e JSON, você será capaz de processar praticamente qualquer dado que quiser analisar. A maioria dos conjuntos de dados online pode ser baixada em um desses formatos, ou em ambos. Ao trabalhar com esses formatos, você também poderá conhecer outros formatos de dados.

No próximo capítulo, escreveremos programas que coletam automaticamente seus próprios dados a partir de fontes online e então criaremos visualizações para esses dados. Essas são habilidades interessantes para ter se você quiser programar como hobby, e são essenciais se estiver interessado em programar profissionalmente.

17
TRABALHANDO COM APIS

Neste capítulo aprenderemos a escrever um programa autocontido para gerar uma visualização baseada em dados recuperados pelo programa. Seu programa usará uma *API* (Application Programming Interface, ou Interface de Programação de Aplicativos) web para solicitar informações específicas de um site automaticamente, em vez de pedir páginas inteiras. Então essas informações serão usadas para gerar uma visualização. Como os programas escritos dessa forma sempre usarão dados atuais para gerar uma visualização, mesmo que esses dados mudem rapidamente, eles estarão sempre atualizados.

Usando uma API web

Uma API web é uma parte de um site projetada para interagir com programas que usam URLs bem específicos a fim de requisitar determinadas informações. Esse tipo de requisição é conhecido como *chamada de API*. Os dados solicitados serão devolvidos em um formato facilmente processável, por exemplo, JSON ou CSV. A maioria das aplicações que depende de fontes de dados externas, como aquelas que se integram a sites de mídias sociais, contam com chamadas de API.

Git e GitHub

Nossa visualização será baseada em informações do GitHub: um site que permite aos programadores colaborar em projetos. Usaremos a API do GitHub para solicitar informações do site sobre projetos Python e então vamos gerar uma visualização interativa da popularidade relativa desses projetos no Pygal.

O GitHub (*https://github.com/*) tem esse nome por causa do Git – um sistema distribuído de controle de versões que permite às equipes de programadores colaborar em projetos. O Git ajuda as pessoas a administrar seus trabalhos individuais em um projeto, de modo que as alterações feitas por uma pessoa não interfiram nas mudanças que outras pessoas estão fazendo. Ao implementar um novo recurso em um projeto, o Git controlará as alterações que você fizer em cada arquivo. Quando seu novo código estiver funcionando, você fará *commit* das alterações feitas e o Git registrará o novo estado de seu projeto. Se você cometer um erro e quiser reverter suas alterações, poderá facilmente voltar para qualquer estado anterior funcional. (Para saber mais sobre controle de versões com o Git, consulte o Apêndice D.) Os projetos no GitHub são armazenados em *repositórios*, que contêm tudo que está associado ao projeto: seu código, informações sobre os colaboradores, qualquer relatório de problema ou bug, e assim por diante.

Quando os usuários do GitHub gostam de um projeto, eles podem lhe conceder uma "estrela" para mostrar seu apoio e monitorar os projetos que queiram usar. Neste capítulo escreveremos um programa para fazer download automático de informações sobre os projetos Python com mais estrelas no GitHub e, em seguida, criaremos uma visualização informativa desses projetos.

Requisitando dados usando uma chamada de API

A API do GitHub permite requisitar várias informações por meio de chamadas de API. Para ver como é a aparência de uma chamada de API, digite o seguinte na barra de endereço de seu navegador e tecle ENTER:

```
https://api.github.com/search/repositories?q=language:python&sort=stars
```

Essa chamada devolve o número de projetos Python hospedados no GitHub no momento, bem como informações sobre os repositórios Python mais populares. Vamos analisar a chamada. A primeira parte, `https://api.github.com/`, direciona a requisição para a parte do site do GitHub que responde a chamadas de API. A próxima parte, `search/repositories`, diz à API para conduzir uma pesquisa em todos os

repositórios do `GitHub.`

O ponto de interrogação depois de `repositories` indica que estamos prestes a passar um argumento. A letra `q` quer dizer query e o sinal de igualdade nos permite começar a especificá-la (`q=`). Ao usar `language:python`, sinalizamos que queremos informações somente sobre os repositórios que tenham Python como a linguagem principal. A última parte, `&sort=stars`, ordena os projetos de acordo com o número de estrelas que receberam.

O trecho a seguir mostra as primeiras linhas da resposta. Ao observá-la, podemos notar que esse URL não tem como propósito ser usado por seres humanos.

```
{
  "total_count": 713062,
  "incomplete_results": false,
  "items": [
    {
      "id": 3544424,
      "name": "httpie",
      "full_name": "jkbrzt/httpie",
      --trecho omitido--
```

Como podemos ver na segunda linha da saída, o GitHub encontrou um total de 713.062 projetos Python na ocasião em que testamos isso. Como o valor de `"incomplete_results"` é `false`, sabemos que a requisição foi bem-sucedida (não está incompleta). Se o GitHub não tivesse sido capaz de processar totalmente a requisição da API, ele teria devolvido `true` aqui. Os `"items"` devolvidos são exibidos na lista que está na sequência, a qual contém detalhes sobre os projetos Python mais populares no GitHub.

Instalando o pacote requests

O pacote `requests` permite que um programa Python solicite facilmente informações a um site e analise a resposta devolvida. Para instalar esse pacote, execute um comando como este:

```
$ pip install --user requests
```

Se você ainda não usou o pip, consulte a seção “Instalando pacotes Python com o pip”. (Talvez você precise usar uma versão um pouco diferente desse comando, conforme a configuração de seu sistema.)

Processando uma resposta de API

Agora começaremos a escrever um programa para fazer uma chamada de API e processar o resultado identificando os projetos Python com mais estrelas no GitHub:

python_repos.py

```
u import requests

  # Faz uma chamada de API e armazena a resposta
v url = 'https://api.github.com/search/repositories?
q=language:python&sort=stars'
w r = requests.get(url)
x print("Status code:", r.status_code)

  # Armazena a resposta da API em uma variável
y response_dict = r.json()

  # Processa o resultado
  print(response_dict.keys())
```

Em u importamos o módulo `requests`. Em v armazenamos o URL da chamada da API e então usamos `requests` para fazer a chamada w. Chamamos `get()`, passamos o URL e armazenamos o objeto com a resposta na variável `r`. O objeto com a resposta tem um atributo chamado `status_code`, que nos informa se a requisição foi bem-sucedida. (Um código de status igual a 200 indica sucesso na resposta.) Em x exibimos o valor de `status_code` para garantir que a chamada foi realizada com sucesso.

A API devolve as informações em formato JSON, portanto usamos o método `json()` y para convertê-las em um dicionário Python. Armazenamos o dicionário resultante em `response_dict`.

Por fim, exibimos as chaves de `response_dict` e vemos o seguinte:

```
Status code: 200
```

```
dict_keys(['items', 'total_count', 'incomplete_results'])
```

Como o código de status é 200, sabemos que a requisição teve sucesso. O dicionário com a resposta contém apenas três chaves: `'items'`, `'total_count'` e `'incomplete_results'`.

NOTA Chamadas simples como essa devem devolver um conjunto completo de resultados, portanto é seguro ignorar o valor associado a `'incomplete_results'`. Porém, quando fizer chamadas de API mais complexas, seu programa deverá conferir esse valor.

Trabalhando com o dicionário de resposta

Agora que temos a informação da chamada de API na forma de um dicionário, podemos trabalhar com os dados armazenados nele. Vamos gerar uma saída que sintetize as informações. Essa é uma boa maneira de garantir que recebemos as informações esperadas e começar a analisar os dados em que estamos interessados:

python_repos.py

```
  import requests

  # Faz uma chamada de API e armazena a resposta
  url = 'https://api.github.com/search/repositories?
  q=language:python&sort=stars'
  r = requests.get(url)
  print("Status code:", r.status_code)

  # Armazena a resposta da API em uma variável
  response_dict = r.json()
u print("Total repositories:", response_dict['total_count'])

  # Explora informações sobre os repositórios
v repo_dicts = response_dict['items']
  print("Repositories returned:", len(repo_dicts))

  # Analisa o primeiro repositório
w repo_dict = repo_dicts[0]
x print("\nKeys:", len(repo_dict))
y for key in sorted(repo_dict.keys()):
```

```
    print(key)
```

Em u exibimos o valor associado a `'total_count'`, que representa o número total de repositórios Python no GitHub.

O valor associado a `'items'` é uma lista que contém vários dicionários, cada um contendo dados sobre um repositório Python individual. Em v armazenamos essa lista de dicionários em `repo_dicts`. Então exibimos o tamanho de `repo_dicts` para ver o número de repositórios para os quais temos informações.

Para observar melhor as informações devolvidas sobre cada repositório, extraímos o primeiro item de `repo_dicts` e o armazenamos em `repo_dict` w. Então exibimos a quantidade de chaves do dicionário para ver quantas informações temos x. Em y exibimos todas as chaves (`keys`) do dicionário para ver quais tipos de informação estão incluídos.

O resultado começa a nos dar uma imagem mais clara dos dados propriamente ditos:

```
  Status code: 200
  Total repositories: 713062
  Repositories returned: 30

u Keys: 68
  archive_url
  assignees_url
  blobs_url
  --trecho omitido--
  url
  watchers
  watchers_count
```

A API do GitHub devolve muitas informações sobre cada repositório: há 68 chaves em `repo_dict` u. Ao observar essas chaves, você terá uma noção do tipo de informação que pode ser extraído a respeito de um projeto. (A única maneira de saber quais informações estão disponíveis por meio de uma API é ler a documentação ou analisar as informações por meio de código, como estamos fazendo nesse caso.)

Vamos extrair os valores de algumas das chaves em `repo_dict`:

python_repos.py

```
  --trecho omitido--
  # Explora informações sobre os repositórios
  repo_dicts = response_dict['items']
  print("Repositories returned:", len(repo_dicts))

  # Analisa o primeiro repositório
  repo_dict = repo_dicts[0]

  print("\nSelected information about first repository:")
u print('Name:', repo_dict['name'])
v print('Owner:', repo_dict['owner']['login'])
w print('Stars:', repo_dict['stargazers_count'])
  print('Repository:', repo_dict['html_url'])
x print('Created:', repo_dict['created_at'])
y print('Updated:', repo_dict['updated_at'])
  print('Description:', repo_dict['description'])
```

Nesse exemplo exibimos os valores de diversas chaves do dicionário do primeiro repositório. Em u exibimos o nome do projeto. Um dicionário completo representa o dono do projeto; assim, em v, usamos a chave `owner` para acessar o dicionário que o representa e então usamos a chave `login` para obter o seu nome de login. Em w exibimos a quantidade de estrelas que o projeto recebeu e o URL do repositório do projeto no GitHub. Em seguida, mostramos a data em que o projeto foi criado x e quando foi atualizado pela última vez y. Por fim, exibimos a descrição do repositório; a saída deve ser semelhante a:

```
Status code: 200
Total repositories: 713065
Repositories returned: 30

Selected information about first repository:
Name: httpie
Owner: jkbrzt
Stars: 16101
Repository: https://github.com/jkbrzt/httpie
Created: 2012-02-25T12:39:13Z
Updated: 2015-07-13T14:56:41Z
```

```
Description: CLI HTTP client; user-friendly cURL replacement featuring
intuitive UI, JSON support, syntax highlighting, wget-like downloads,
extensions, etc.
```

Podemos ver que o projeto Python com mais estrelas no GitHub (na época em que esta obra foi escrita) é o HTTPie, cujo proprietário é o usuário *jkbrzt*, e recebeu estrelas de mais de 16 mil usuários do GitHub. Vemos também o URL do repositório do projeto, a data de sua criação – fevereiro de 2012 – e que o projeto foi atualizado recentemente. Por fim, a descrição nos informa que o HTTPie ajuda a fazer chamadas HTTP a partir de um terminal (*CLI* é a abreviatura de *command line interface*, ou interface de linha de comando).

Resumo dos principais repositórios

Quando criarmos uma visualização para esses dados, vamos querer incluir mais de um repositório. Escreveremos um laço para exibir informações selecionadas sobre cada um dos repositórios devolvidos pela chamada de API para que possamos incluir todos eles na visualização.

python_repos.py

```
--trecho omitido--
# Explora informações sobre os repositórios
repo_dicts = response_dict['items']
print("Repositories returned:", len(repo_dicts))

u print("\nSelected information about each repository:")
v for repo_dict in repo_dicts:
      print('\nName:', repo_dict['name'])
      print('Owner:', repo_dict['owner']['login'])
      print('Stars:', repo_dict['stargazers_count'])
      print('Repository:', repo_dict['html_url'])
      print('Description:', repo_dict['description'])
```

Exibimos uma mensagem introdutória em u. Em v percorremos todos os dicionários em `repo_dicts` com um laço. Nesse laço exibimos o nome de cada projeto, o seu proprietário, quantas estrelas o projeto recebeu, seu URL no GitHub e a sua descrição:

```
Status code: 200
Total repositories: 713067
Repositories returned: 30

Selected information about each repository:

Name: httpie
Owner: jkbrzt
Stars: 16101
Repository: https://github.com/jkbrzt/httpie
Description: CLI HTTP client; user-friendly cURL replacement featuring
intuitive UI, JSON support, syntax highlighting, wget-like downloads,
extensions, etc.

Name: django
Owner: django
Stars: 15028
Repository: https://github.com/django/django
Description: The Web framework for perfectionists with deadlines.
--trecho omitido--

Name: powerline
Owner: powerline
Stars: 4315
Repository: https://github.com/powerline/powerline
Description: Powerline is a statusline plugin for vim, and provides
statuslines and prompts for several other applications, including zsh,
bash, tmux, IPython, Awesome and Qtile.
```

Alguns projetos interessantes aparecem nesse resultado, e pode valer a pena dar uma olhada em alguns. No entanto, não gaste muito tempo nisso, pois estamos prestes a criar uma visualização que facilitará bastante a leitura dos resultados.

Monitorando os limites da taxa de uso da API

A maioria das APIs tem uma taxa de uso limitada, o que significa que há um limite para quantas requisições podemos fazer em determinado período de tempo. Para ver se estamos nos aproximando dos limites do GitHub, forneça o endereço *https://api.github.com/rate_limit* em um navegador web.

Você deverá ver uma resposta como esta:

```
  {
    "resources": {
      "core": {
        "limit": 60,
        "remaining": 58,
        "reset": 1426082320
      },
u     "search": {
v       "limit": 10,
w       "remaining": 8,
x       "reset": 1426078803
      }
    },
    "rate": {
      "limit": 60,
      "remaining": 58,
      "reset": 1426082320
    }
  }
```

A informação em que estamos interessados é o limite da taxa de uso da API de pesquisa u. Em v vemos que o limite é de dez requisições por minuto e que temos oito requisições restantes para o minuto atual w. O valor de reinicialização (reset) representa o instante na *Era Unix* (Unix time) ou em *epoch time* (o número de segundos desde a meia-noite de 1 de janeiro de 1970) em que nossa quota será reiniciada x. Se atingir sua quota, você obterá uma resposta breve que permitirá saber que o limite da API foi atingido. Se alcançar o limite, basta esperar até que sua quota seja reiniciada.

NOTA Muitas APIs exigem que você se registre e obtenha uma chave para fazer chamadas de API. Na época em que este texto foi escrito, o GitHub não tinha esse requisito, porém, se você adquirir uma chave de API, seus limites serão bem maiores.

Visualizando os repositórios usando o Pygal

Agora que temos alguns dados interessantes, vamos criar uma visualização

que mostre a popularidade relativa dos projetos Python no GitHub. Criaremos um gráfico de barras interativo: a altura de cada barra representará o número de estrelas que o projeto recebeu. Clicar em uma barra levará você para a página inicial do projeto no GitHub. A seguir, temos uma tentativa inicial:

python_repos.py

```
  import requests
  import pygal
  from pygal.style import LightColorizedStyle as LCS, LightenStyle as LS

  # Faz uma chamada de API e armazena a resposta
  URL = 'https://api.github.com/search/repositories?
  q=language:python&sort=star'
  r = requests.get(URL)
  print("Status code:", r.status_code)

  # Armazena a resposta da API em uma variável
  response_dict = r.json()
  print("Total repositories:", response_dict['total_count'])

  # Explora informações sobre os repositórios
  repo_dicts = response_dict['items']

u names, stars = [], []
  for repo_dict in repo_dicts:
v     names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

  # Cria a visualização
w my_style = LS('#333366', base_style=LCS)
x chart = pygal.Bar(style=my_style, x_label_rotation=45,
show_legend=False)
  chart.title = 'Most-Starred Python Projects on GitHub'
  chart.x_labels = names

y chart.add('', stars)
  chart.render_to_file('python_repos.svg')
```

Começamos importando o `pygal` e os estilos do Pygal de que

precisaremos para o gráfico. Continuamos exibindo o status da resposta da chamada à API e o número total de repositórios encontrados para que possamos saber caso haja algum problema com a chamada da API. Não exibimos mais as informações sobre os projetos específicos devolvidos, pois elas serão incluídas na visualização.

Em u criamos duas listas vazias para armazenar os dados que incluiremos no gráfico. Precisaremos do nome de cada projeto para rotular as barras e do número de estrelas para determinar a altura delas. No laço, concatenamos nessas listas o nome de cada projeto e o número de estrelas que ele tem v.

Em seguida, definimos um estilo usando a classe `LightenStyle` (alias `LS`) e usamos um tom de azul-escuro como base w. Também passamos o argumento `base_style` para utilizar a classe `LightColorizedStyle` (alias `LCS`). Então usamos `Bar()` para criar um gráfico de barras simples e lhe passamos `my_style` x. Além disso, passamos outros dois argumentos de estilo: definimos a rotação dos nomes ao longo do eixo x em 45 graus (`x_label_rotation=45`) e ocultamos a legenda, pois estamos plotando apenas uma série no gráfico (`show_legend=False`). Então fornecemos um título ao gráfico e definimos o atributo `x_labels` com a lista `names`.

Como não é necessário nomear essa série de dados, passamos uma string vazia para o rótulo quando adicionamos os dados y. O gráfico resultante pode ser visto na Figura 17.1. Podemos ver que os primeiros projetos são significativamente mais populares que os demais, mas todos eles são importantes no ecossistema de Python.

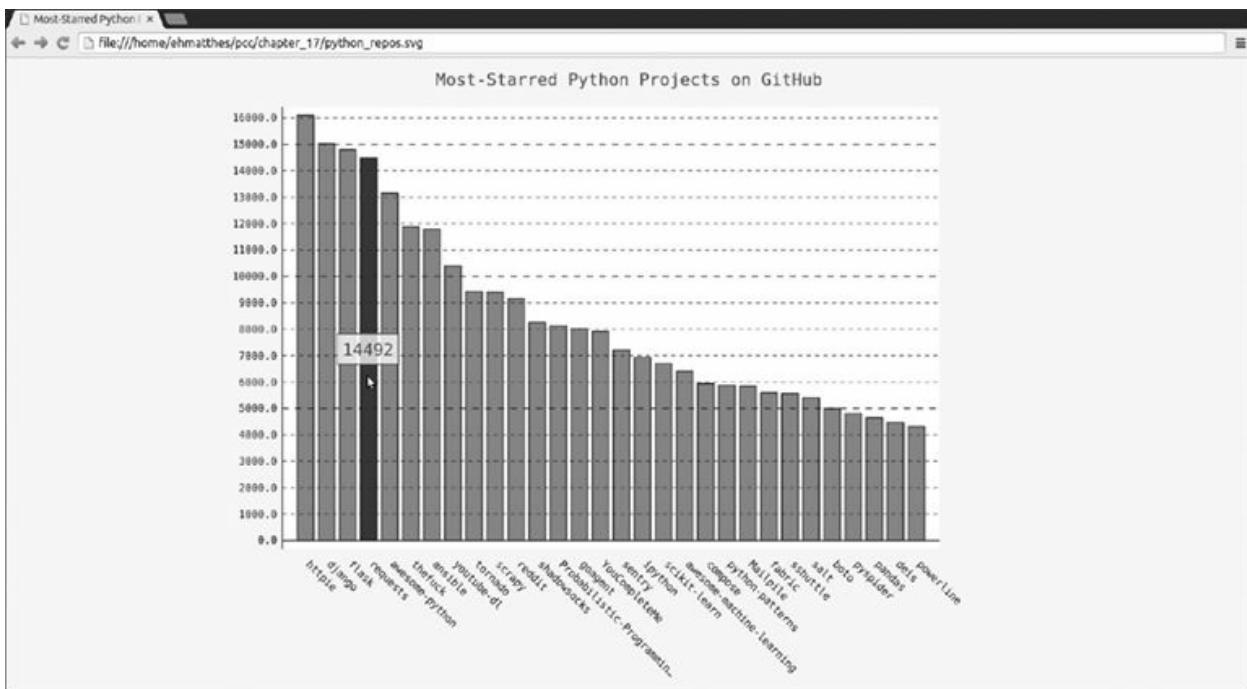



Figura 17.1 – Os projetos Python com mais estrelas no GitHub.

Aperfeiçoando os gráficos do Pygal

Vamos melhorar a estilização de nosso gráfico. Faremos algumas personalizações diferentes, portanto, em primeiro lugar, reestruture um pouco o código criando um objeto de configuração que contenha todas as nossas personalizações, para que seja passado para `Bar()`:

python_repos.py

```
  --trecho omitido--

  # Cria a visualização
  my_style = LS('#333366', base_style=LCS)

u my_config = pygal.Config()
v my_config.x_label_rotation = 45
  my_config.show_legend = False
w my_config.title_font_size = 24
  my_config.label_font_size = 14
  my_config.major_label_font_size = 18
x my_config.truncate_label = 15
y my_config.show_y_guides = False
```

```
z my_config.width = 1000

{ chart = pygal.Bar(my_config, style=my_style)
  chart.title = 'Most-Starred Python Projects on GitHub'
  chart.x_labels = names

  chart.add('', stars)
  chart.render_to_file('python_repos.svg')
```

Em u criamos uma instância chamada `my_config` da classe `Config` do Pygal; modificar os atributos de `my_config` personalizará a aparência do gráfico. Definimos os dois atributos `x_label_rotation` e `show_legend` v, originalmente passados como argumentos nomeados quando criamos uma instância de `Bar`. Em w definimos o tamanho da fonte para o título do gráfico e para os rótulos menores e maiores. Os rótulos menores nesse gráfico são os nomes dos projetos ao longo do eixo x e a maior parte dos números no eixo y. Os rótulos maiores são apenas os rótulos do eixo y que marcam incrementos de 5.000 estrelas. Esses rótulos serão maiores, e é por isso que os diferenciamos. Em x usamos `truncate_label` para reduzir os nomes de projeto mais longos a 15 caracteres. (Quando você passar o mouse sobre um nome de projeto truncado na tela, o nome completo aparecerá.) Em seguida, ocultamos as linhas horizontais do gráfico definindo `show_y_guides` com `False` y. Por fim, em z, definimos uma largura personalizada para que o gráfico use mais do espaço disponível no navegador.

Agora, quando criamos uma instância de `Bar` at {, passamos `my_config` como primeiro argumento, e todas as nossas definições de configuração serão enviadas em um só argumento. Podemos fazer quantas modificações de estilo e de configuração quisermos por meio de `my_config`, e a linha em { não mudará. A Figura 17.2 mostra o gráfico reestilizado.

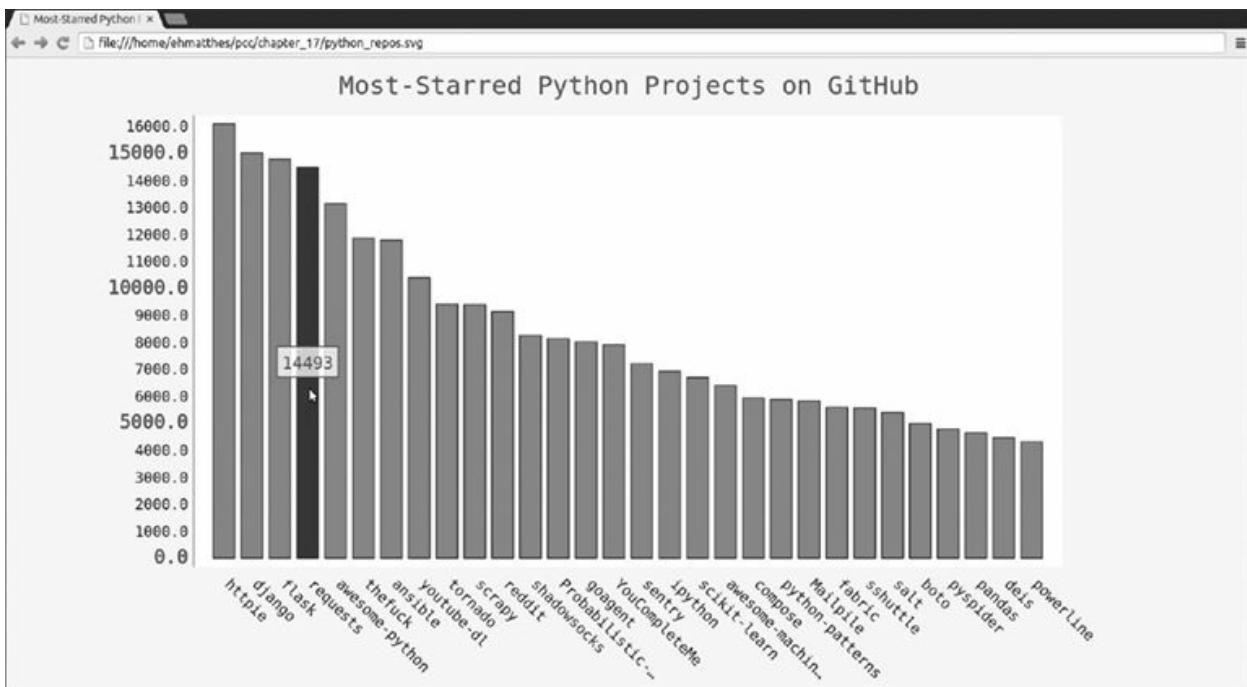



Figura 17.2 – A estilização do gráfico foi melhorada.

Acrescentando dicas de contexto personalizadas

No Pygal, passar o cursor sobre uma barra individual faz com que as informações representadas pela barra sejam exibidas. Elas são comumente chamadas de *dicas de contexto* (tooltips) e, nesse caso, mostram o número de estrelas que um projeto tem. Criaremos uma dica de contexto personalizada que mostre também a descrição de cada projeto.

Vamos ver um pequeno exemplo que usa os três primeiros projetos plotados individualmente, com rótulos personalizados passados para cada barra. Para isso passaremos uma lista de dicionários para `add()` no lugar de uma lista de valores:

bar_descriptions.py

```
import pygal
from pygal.style import LightColorizedStyle as LCS, LightenStyle as LS

my_style = LS('#333366', base_style=LCS)
chart = pygal.Bar(style=my_style, x_label_rotation=45,
show_legend=False)
```

```
  chart.title = 'Python Projects'
  chart.x_labels = ['httpie', 'django', 'flask']

u plot_dicts = [
v     {'value': 16101, 'label': 'Description of httpie.'},
      {'value': 15028, 'label': 'Description of django.'},
      {'value': 14798, 'label': 'Description of flask.'},
      ]

w chart.add('', plot_dicts)
  chart.render_to_file('bar_descriptions.svg')
```

Em u definimos uma lista chamada `plot_dicts`, que contém três dicionários: um para o projeto HTTPie, um para o projeto Django e outro para o Flask. Cada dicionário tem duas chaves: `'value'` e `'label'`. O Pygal usa o número associado a `'value'` para descobrir a altura que cada barra deve ter, e utiliza a string associada a `'label'` para criar a dica de contexto de cada barra. Por exemplo, o primeiro dicionário em v criará uma barra que representa um projeto com 16.101 estrelas, e sua dica de contexto conterá *Description of httpie* (Descrição de httpie).

O método `add()` precisa de uma string e de uma lista. Quando chamamos esse método, passamos a lista de dicionários que representa as barras (`plot_dicts`) w. A Figura 17.3 mostra uma das dicas de contexto. O Pygal inclui o número de estrelas como uma dica de contexto default, além da dica de contexto personalizada que lhe passamos.

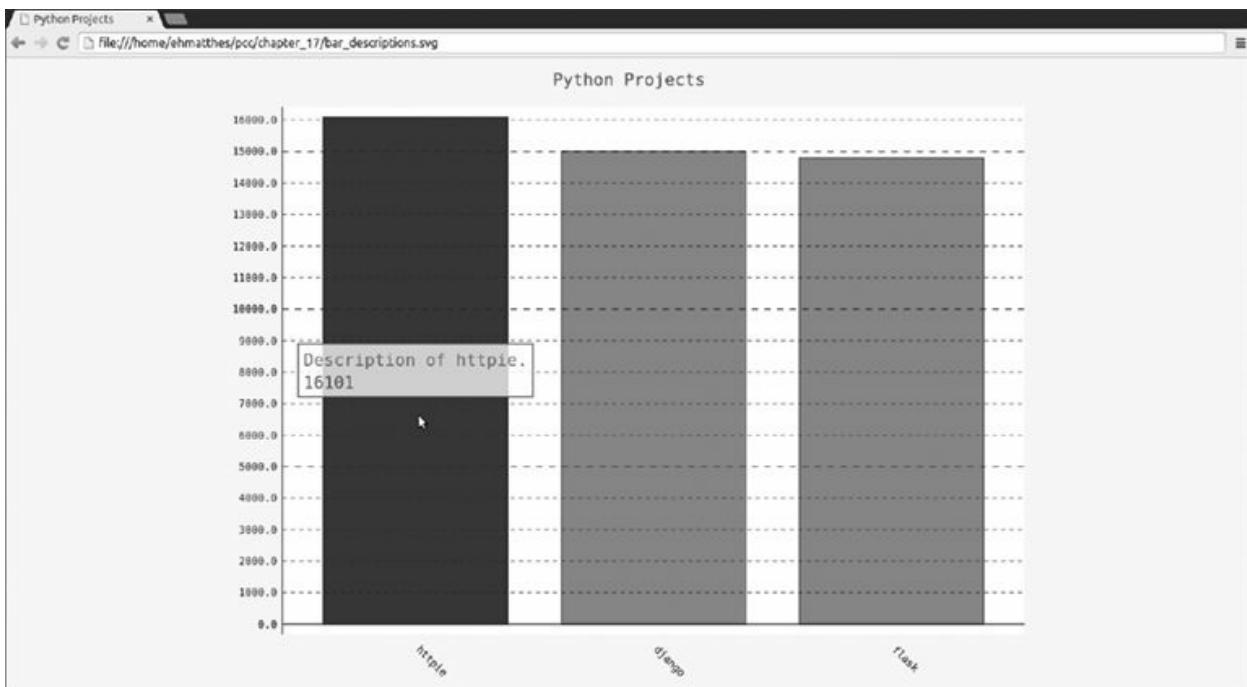



Figura 17.3 – Cada barra tem uma dica de contexto personalizada.

Plotando os dados

Para plotar nossos dados, vamos gerar `plot_dicts` automaticamente para os 30 projetos devolvidos pela chamada de API.

Eis o código para fazer isso:

python_repos.py

```
  --trecho omitido--
  # Explora informações sobre os repositórios
  repo_dicts = response_dict['items']
  print("Number of items:", len(repo_dicts))

u names, plot_dicts = [], []
  for repo_dict in repo_dicts:
      names.append(repo_dict['name'])

v     plot_dict = {
          'value': repo_dict['stargazers_count'],
          'label': repo_dict['description'],
          }
w     plot_dicts.append(plot_dict)
```

```
  # Cria a visualização
  my_style = LS('#333366', base_style=LCS)
  --trecho omitido--

x chart.add('', plot_dicts)
  chart.render_to_file('python_repos.svg')
```

Em u criamos uma lista vazia para `names` e outra para `plot_dicts`. Ainda precisamos da lista `names` para gerar os rótulos do eixo x.

No laço criamos o dicionário `plot_dict` para cada projeto v. Armazenamos o número de estrelas com a chave `'value'` e a descrição do projeto com a chave `'label'` em cada `plot_dict`. Então concatenamos o `plot_dict` de cada projeto em `plot_dicts` w. Em x passamos a lista `plot_dicts` para `add()`. A Figura 17.4 mostra o gráfico resultante.

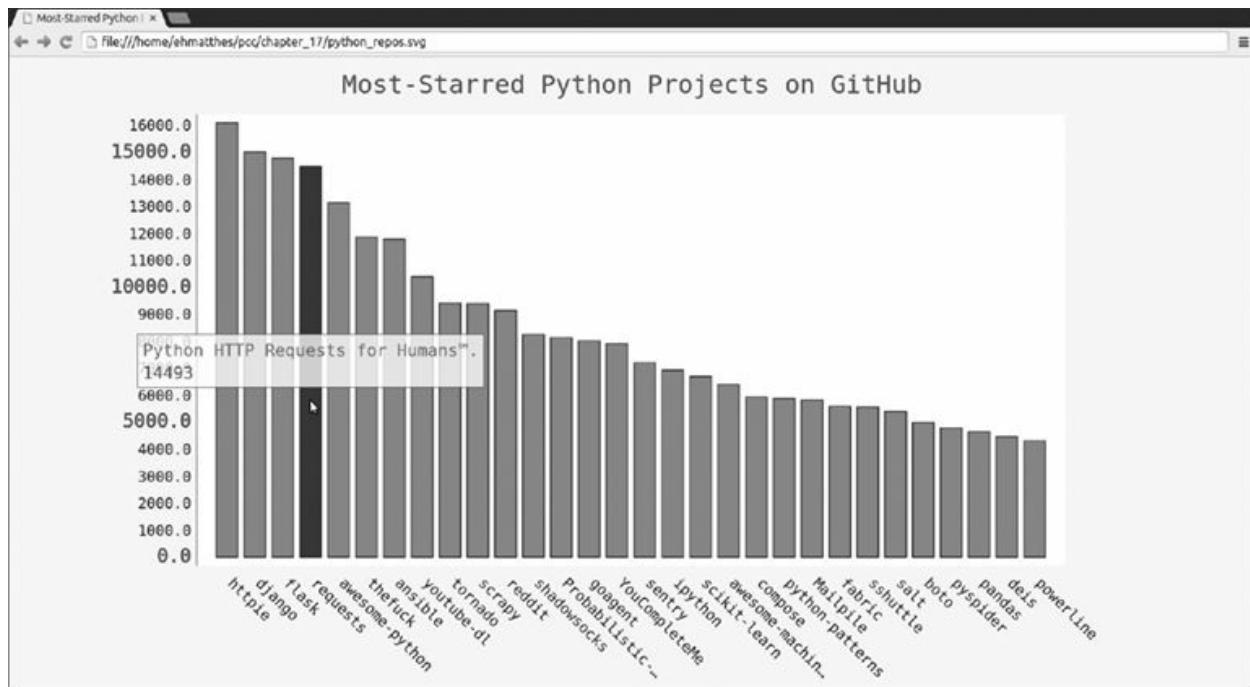



Figura 17.4 – Passar o mouse sobre uma barra mostra a descrição do projeto.

Adicionando links que podem ser clicados em nosso gráfico

O Pygal também permite usar cada barra do gráfico como um link para um site. Para acrescentar essa funcionalidade, basta adicionar uma linha em nosso código, tirando proveito do dicionário que criamos para cada

projeto. Adicionamos um novo par chave-valor ao `plot_dict` de cada projeto usando a chave `'xlink'`:

python_repos.py

```
--trecho omitido--
names, plot_dicts = [], []
for repo_dict in repo_dicts:
    names.append(repo_dict['name'])

    plot_dict = {
        'value': repo_dict['stargazers_count'],
        'label': repo_dict['description'],
        'xlink': repo_dict['html_url'],
        }
    plot_dicts.append(plot_dict)
--trecho omitido--
```

O Pygal usa o URL associado a `'xlink'` para transformar cada barra em um link ativo. Você pode clicar em qualquer uma das barras do gráfico e a página desse projeto no GitHub será automaticamente aberta em uma nova aba em seu navegador. Agora você tem uma visualização informativa e interativa dos dados obtidos por meio de uma API!

A API de Hacker News

Para explorar o uso de chamadas de API em outros sites, daremos uma olhada em Hacker News (*http://news.ycombinator.com/*). No Hacker News, as pessoas compartilham artigos sobre programação e tecnologia, e se envolvem em discussões entusiasmadas sobre esses artigos. A API do Hacker News oferece acesso a dados sobre todos os artigos submetidos e os comentários do site, disponíveis sem a necessidade de se registrar para obter uma chave.

A chamada a seguir devolve informações sobre os principais artigos do momento (na ocasião em que este livro foi escrito):

```
https://hacker-news.firebaseio.com/v0/item/9884165.json
```

A resposta é um dicionário com informações sobre o artigo cujo ID é

9884165:

```
  {
u     'url': 'http://www.bbc.co.uk/news/science-environment-33524589',
      'type': 'story',
v     'title': 'New Horizons: Nasa spacecraft speeds past Pluto',
w     'descendants': 141,
      'score': 230,
      'time': 1436875181,
      'text': '',
      'by': 'nns',
      'id': 9884165,
x     'kids': [9884723, 9885099, 9884789, 9885604, 9885844]
  }
```

O dicionário contém várias chaves com as quais podemos trabalhar, por exemplo, `'url'` u e `'title'` v. A chave `'descendants'` contém a quantidade de comentários que um artigo recebeu w. A chave `'kids'` oferece os IDs de todos os comentários feitos diretamente em resposta a esse artigo submetido x. Cada um desses comentários pode ter filhos próprios também, portanto o número de descendentes que um artigo submetido tem pode ser maior que o número de seus filhos.

Vamos fazer uma chamada de API que devolva os IDs dos principais artigos do momento no Hacker News e então analisaremos cada um desses artigos:

hn_submissions.py

```
  import requests

  from operator import itemgetter

  # Faz uma chamada de API e armazena a resposta
u url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
  r = requests.get(url)
  print("Status code:", r.status_code)

  # Processa informações sobre cada artigo submetido
v submission_ids = r.json()
w submission_dicts = []
```

```
  for submission_id in submission_ids[:30]:
      # Cria uma chamada de API separada para cada artigo submetido
x     url = ('https://hacker-news.firebaseio.com/v0/item/' +
              str(submission_id) + '.json')
      submission_r = requests.get(url)
      print(submission_r.status_code)
      response_dict = submission_r.json()

y     submission_dict = {
          'title': response_dict['title'],
          'link': 'http://news.ycombinator.com/item?id=' +
  str(submission_id),
z         'comments': response_dict.get('descendants', 0)
          }
      submission_dicts.append(submission_dict)

{ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                              reverse=True)

| for submission_dict in submission_dicts:
      print("\nTitle:", submission_dict['title'])
      print("Discussion link:", submission_dict['link'])
      print("Comments:", submission_dict['comments'])
```

Inicialmente fizemos a chamada de API e exibimos o status da resposta u. Essa chamada de API devolve uma lista contendo os IDs dos 500 artigos mais populares do Hacker News no momento em que a chamada foi feita. Então convertemos o texto da resposta em uma lista Python em v, que armazenamos em `submission_ids`. Usaremos esses IDs para criar um conjunto de dicionários em que cada um armazenará informações sobre um dos artigos submetidos.

Criamos uma lista vazia chamada `submission_dicts` em w para armazenar esses dicionários. Então percorremos os IDs dos 30 principais artigos submetidos com um laço. Fazemos uma nova chamada de API para cada artigo gerando um URL que inclui o valor atual de `submission_id` x. Exibimos o status de cada requisição para que possamos ver se ela foi bem-sucedida.

Em y criamos um dicionário para o artigo submetido processado no

momento, no qual armazenamos o título do artigo e um link para a página de discussão desse item. Em z armazenamos o número de comentários no dicionário. Se um artigo ainda não teve nenhum comentário, a chave 'descendants' não estará presente. Quando você não tiver certeza de que uma chave existe em um dicionário, utilize o método dict.get(), que devolve o valor associado à chave especificada se ela existir, ou o valor que você fornecer se ela não existir (0 nesse exemplo). Por fim, concatenamos cada submission_dict à lista submission_dicts.

Os artigos submetidos no Hacker News são classificados de acordo com uma pontuação geral, baseada em vários fatores, incluindo quantos votos receberam, quantos comentários foram feitos e quão recentemente o artigo foi submetido. Queremos ordenar a lista de dicionários de acordo com o número de comentários. Para isso, usamos uma função chamada itemgetter() {, proveniente do módulo operator. Passamos a chave 'comments' a essa função e ela extrai o valor associado a essa chave de cada dicionário da lista. A função sorted() então utiliza esse valor como base para ordenar a lista. Ordenamos a lista na ordem inversa para colocar as histórias mais comentadas antes.

Depois que a lista estiver ordenada, nós a percorremos com um laço em | e exibimos três informações sobre cada um dos principais artigos submetidos: o título, um link para a página de discussão e o número de comentários que o artigo submetido tem no momento:

```
Status code: 200
200
200
200
--trecho omitido--

Title: Firefox deactivates Flash by default
Discussion link: http://news.ycombinator.com/item?id=9883246
Comments: 231

Title: New Horizons: Nasa spacecraft speeds past Pluto
Discussion link: http://news.ycombinator.com/item?id=9884165
Comments: 142
```

```
Title: Iran Nuclear Deal Is Reached With World Powers
Discussion link: http://news.ycombinator.com/item?id=9884005
Comments: 141

Title: Match Group Buys PlentyOfFish for $575M
Discussion link: http://news.ycombinator.com/item?id=9884417
Comments: 75

Title: Our Nexus 4 devices are about to explode
Discussion link: http://news.ycombinator.com/item?id=9885625
Comments: 14

--trecho omitido--
```

Você poderia usar um processo semelhante para acessar e analisar informações com qualquer API. Com esses dados você poderia criar uma visualização que mostre quais artigos submetidos inspiraram as discussões recentes mais entusiasmadas.

FAÇA VOCÊ MESMO

17.1 – Outras linguagens: Modifique a chamada de API em *python_repos.py* para que ela gere um gráfico mostrando os projetos mais populares em outras linguagens. Experimente usar linguagens como *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell* e *Go*.

17.2 – Discussões entusiasmadas: Usando os dados de *hn_submissions.py*, crie um gráfico de barras que mostre as discussões mais entusiasmadas do momento no Hacker News. A altura de cada barra deve corresponder ao número de comentários que cada artigo submetido tem. O rótulo de cada barra deve incluir o título do artigo submetido, e cada barra deve atuar como um link para a página de discussão desse artigo.

17.3 – Testando python_repos.py: Em *python_repos.py*, exibimos o valor de `status_code` para garantir que a chamada de API foi bem-sucedida. Escreva um programa chamado *test_python_repos.py* que use `unittest` para conferir se o valor de `status_code` é 200. Descubra outras asserções que você possa fazer – por exemplo, se o número de itens devolvidos é o que se espera e se o número total de repositórios é maior que uma determinada quantidade.

Resumo

Neste capítulo aprendemos a usar APIs para escrever programas autocontidos que coletem automaticamente os dados necessários e usem esses dados para criar uma visualização. Usamos a API do GitHub para explorar os projetos Python com mais estrelas no GitHub e vimos rapidamente a API do Hacker News também. Aprendemos a usar o pacote `requests` para fazer uma chamada de API ao GitHub de modo automático e a processar os resultados dessa chamada. Também apresentamos algumas configurações do Pygal para personalizar melhor a aparência dos gráficos que você gerar.

No último projeto, usaremos Django para criar uma aplicação web.

PROJETO 3

APLICAÇÕES WEB

18
INTRODUÇÃO AO DJANGO

Internamente, os sites de hoje, na verdade, são aplicações sofisticadas que agem como aplicações desktop completas. Python tem um ótimo conjunto de ferramentas para criar aplicações web. Neste capítulo aprenderemos a usar Django (*http://djangoproject.com/*) para criar um projeto chamado Learning Log (Registro de aprendizado) – um sistema de diário online que permite manter o controle de informações que você adquiriu sobre determinados assuntos.

Escreveremos uma especificação para esse projeto e, em seguida, definiremos modelos para os dados com os quais a aplicação trabalhará. Usaremos o sistema de administração de Django para inserir alguns dados iniciais e então aprenderemos a escrever views e templates para que esse framework possa criar as páginas de seu site.

O Django é um *framework web* – um conjunto de ferramentas projetado para ajudar você a criar sites interativos. Esse framework é capaz de responder a requisições de páginas e facilita ler e escrever em um banco de dados, administrar usuários e outras tarefas. Nos Capítulos 19 e 20 aperfeiçoaremos o projeto Learning Log e o implantaremos em um servidor ativo para que você possa usá-lo (e seus amigos também).

Criando um projeto

Ao começar um projeto, você deve inicialmente descrevê-lo em uma especificação (*spec*). Então você deve configurar um ambiente virtual em que o projeto será criado.

Escrevendo uma especificação

Uma especificação completa detalha os objetivos do projeto, descreve suas funcionalidades e discute sua aparência e a interface de usuário. Como

qualquer bom projeto ou plano de negócios, uma especificação deve permitir que você mantenha o foco e conduza o seu projeto no caminho certo. Não redigiremos aqui uma especificação de projeto completa, mas estabeleceremos algumas metas claras para que o foco de nosso processo de desenvolvimento seja mantido. Eis a especificação que usaremos:

> *Criaremos uma aplicação web chamada Learning Log que permite aos usuários registrar os assuntos em que estiverem interessados e criar entradas de diário à medida que aprenderem algo sobre cada assunto. A página inicial de Learning Log deve descrever o site e convidar os usuários a se cadastrar ou a fazer login. Depois que estiver logado, um usuário deve ser capaz de criar novos assuntos e adicionar novas entradas, além de ler e editar entradas existentes.*

Quando aprender algo sobre um novo assunto, manter um diário do que você aprendeu pode ser útil para controlar e rever informações. Uma boa aplicação pode tornar esse processo eficiente.

Criando um ambiente virtual

Para trabalhar com Django, vamos inicialmente criar um ambiente virtual em que trabalharemos. Um *ambiente virtual* é um local de seu sistema em que você pode instalar pacotes e isolá-los de todos os demais pacotes Python. Separar as bibliotecas de um projeto das bibliotecas de outros projetos é vantajoso e será necessário quando implantarmos Learning Log em um servidor no Capítulo 20.

Crie um novo diretório chamado *learning_log* para o seu projeto, navegue até esse diretório em um terminal e crie um ambiente virtual. Se você usa Python 3, deverá ser capaz de criar um ambiente virtual com o comando a seguir:

```
learning_log$ python -m venv ll_env
learning_log$
```

Nesse caso estamos executando o módulo `venv` e usando-o para criar um ambiente virtual chamado *ll_env*. Se isso funcionar, vá para a seção "Ativando o ambiente virtual". Se não funcionar, leia a próxima seção, "Instalando o virtualenv".

Instalando o virtualenv

Se você usa uma versão mais antiga de Python ou se seu sistema não estiver configurado para usar o módulo `venv` corretamente, instale o pacote virtualenv. Para instalá-lo, execute o seguinte:

```
$ pip install --user virtualenv
```

Tenha em mente que você talvez precise usar uma versão um pouco diferente desse comando. (Se você ainda não usou o pip, consulte a seção "Instalando pacotes Python com o pip".)

NOTA Se você usa Linux e esse comando não funcionou, poderá instalar o virtualenv por meio do gerenciador de pacotes de seu sistema. No Ubuntu, por exemplo, o comando `sudo apt-get install python-virtualenv` instalará o virtualenv.

Vá para o diretório *learning_log* em um terminal e crie um ambiente virtual, assim:

```
learning_log$ virtualenv ll_env
New python executable in ll_env/bin/python
Installing setuptools, pip...done.
learning_log$
```

NOTA Se você tem mais de uma versão de Python instalada em seu sistema, especifique a versão do virtualenv a ser usada. Por exemplo, o comando `virtualenv ll_env --python=python3` criará um ambiente virtual que utiliza Python 3.

Ativando o ambiente virtual

Agora que temos um ambiente virtual definido, é necessário ativá-lo com o seguinte comando:

```
  learning_log$ source ll_env/bin/activate
u (ll_env)learning_log$
```

Esse comando executa o script *activate* em *ll_env/bin*. Quando o ambiente estiver ativo, você verá o nome dele entre parênteses, como vemos em u; então você poderá instalar pacotes no ambiente e usar pacotes que já tenham sido instalados. Os pacotes que você instalar em *ll_env* estarão disponíveis somente enquanto o ambiente estiver ativo.

NOTA Se você usa Windows, utilize o comando `ll_env\Scripts\activate` (sem a palavra `source`) para ativar o ambiente virtual.

Para interromper um ambiente virtual, digite `deactivate`:

```
(ll_env)learning_log$ deactivate
learning_log$
```

O ambiente também se tornará inativo se você fechar o terminal em que ele estiver executando.

Instalando o Django

Depois de ter criado e ativado o seu ambiente virtual, instale o Django:

```
(ll_env)learning_log$ pip install Django
Installing collected packages: Django
Successfully installed Django
Cleaning up...
(ll_env)learning_log$
```

Como estamos trabalhando em um ambiente virtual, esse comando é o mesmo em todos os sistemas. Não há necessidade de usar a flag `--user` nem de usar comandos mais longos como `python -m pip install` *`nome_do_pacote`*.

Tenha em mente que o Django estará disponível somente quando o ambiente estiver ativo.

Criando um projeto em Django

Sem sair do ambiente virtual ativo (lembre-se de verificar se *ll_env* está entre parênteses), execute os comandos a seguir para criar um novo projeto:

```
u (ll_env)learning_log$ django-admin.py startproject learning_log .
v (ll_env)learning_log$ ls
  learning_log ll_env manage.py
w (ll_env)learning_log$ ls learning_log
  __init__.py  settings.py  urls.py  wsgi.py
```

O comando em **u** diz a Django para criar um novo projeto chamado *learning_log*. O ponto no final do comando cria o novo projeto com uma estrutura de diretórios que facilitará a implantação da aplicação em um

servidor quando terminarmos o seu desenvolvimento.

NOTA Lembre-se desse ponto; caso contrário, você poderá se deparar com alguns problemas de configuração quando implantarmos a aplicação. Se você se esquecer do ponto, apague os arquivos e as pastas criados (exceto *ll_env*) e execute o comando novamente.

A execução do comando `ls` (`dir` no Windows) **v** mostra que Django criou um novo diretório chamado *learning_log*. Um arquivo chamado *manage.py* também foi criado: é um pequeno programa que aceita comandos e os passa para a parte relevante de Django que os executa. Usaremos esses comandos para administrar tarefas como trabalhar com bancos de dados e executar servidores.

O diretório *learning_log* contém quatro arquivos **w**, entre os quais os mais importantes são *settings.py*, *urls.py* e *wsgi.py*. O arquivo *settings.py* controla o modo como Django interage com o seu sistema e administra o seu projeto. Modificaremos algumas dessas configurações e acrescentaremos outras configurações próprias à medida que o projeto evoluir. O arquivo *urls.py* informa a Django quais páginas devem ser criadas em resposta a requisições do navegador. O arquivo *wsgi.py* ajuda Django a servir os arquivos que ele criar. O nome do arquivo é um acrônimo para *web server gateway interface* (interface de gateway do servidor web).

Criando o banco de dados

Como o Django armazena a maior parte das informações relacionadas a um projeto em um banco de dados, precisamos criar um para que o framework possa trabalhar com ele. Para criar o banco de dados do projeto Learning Log, digite o comando a seguir (ainda no ambiente ativo):

```
  (ll_env)learning_log$ python manage.py migrate
u Operations to perform:
    Synchronize unmigrated apps: messages, staticfiles
    Apply all migrations: contenttypes, sessions, auth, admin
    --trecho omitido--
    Applying sessions.0001_initial... OK
v (ll_env)learning_log$ ls
  db.sqlite3  learning_log  ll_env  manage.py
```

Sempre que modificamos um banco de dados, dizemos que estamos *migrando* o banco de dados. Executar o comando `migrate` pela primeira vez informa a Django para garantir que o banco de dados esteja de acordo com o estado atual do projeto. Na primeira vez que executamos esse comando em um novo projeto que use SQLite (mais sobre SQLite em breve), o Django criará um novo banco de dados para nós. Em u o Django informa que criará as tabelas do banco de dados necessárias para armazenar as informações que usaremos nesse projeto (*Synchronize unmigrated apps*, ou Sincroniza as aplicações não migradas) e então garante que a estrutura do banco de dados esteja de acordo com o código atual (*Apply all migrations*, ou Aplica todas as migrações).

A execução do comando `ls` mostra que Django criou outro arquivo chamado *db.sqlite3* v. O SQLite é um banco de dados que executa com base em um único arquivo; é ideal para escrever aplicações simples, pois você não precisará prestar muita atenção no gerenciamento do banco de dados.

Visualizando o projeto

Vamos garantir que Django configurou o projeto de modo apropriado. Execute o comando `runserver` da seguinte maneira:

```
  (ll_env)learning_log$ python manage.py runserver
  Performing system checks...

u System check identified no issues (0 silenced).
  July 15, 2015 - 06:23:51
v Django version 1.8.4, using settings 'learning_log.settings'
w Starting development server at http://127.0.0.1:8000/
  Quit the server with CONTROL-C.
```

O Django inicia um servidor para que você possa visualizar o projeto em seu sistema e ver como ele funciona. Quando você solicitar uma página fornecendo um URL em um navegador, o servidor Django responderá a essa requisição construindo a página apropriada e enviando-a para o navegador.

Em u Django verifica se o projeto está configurado de modo apropriado; em v a versão de Django em uso e o nome do arquivo de configurações utilizado são informados; em w o URL em que o projeto está sendo servido é apresentado. O URL *http://127.0.0.1:8000/* informa que o projeto está ouvindo requisições na porta 8000 de seu computador – chamada de localhost. O termo *localhost* se refere a um servidor que processa requisições somente em seu sistema; ele não permite que outras pessoas vejam as páginas que você está desenvolvendo.

Agora abra um navegador web e forneça o URL *http://localhost:8000/*, ou *http://127.0.0.1:8000/* se o primeiro não funcionar. Você deverá ver algo como o que está na Figura 18.1: uma página criada por Django para que você saiba que tudo está funcionando adequadamente até agora. Mantenha o servidor executando por enquanto, mas se quiser interrompê-lo, poderá fazer isso pressionando CTRL-C.

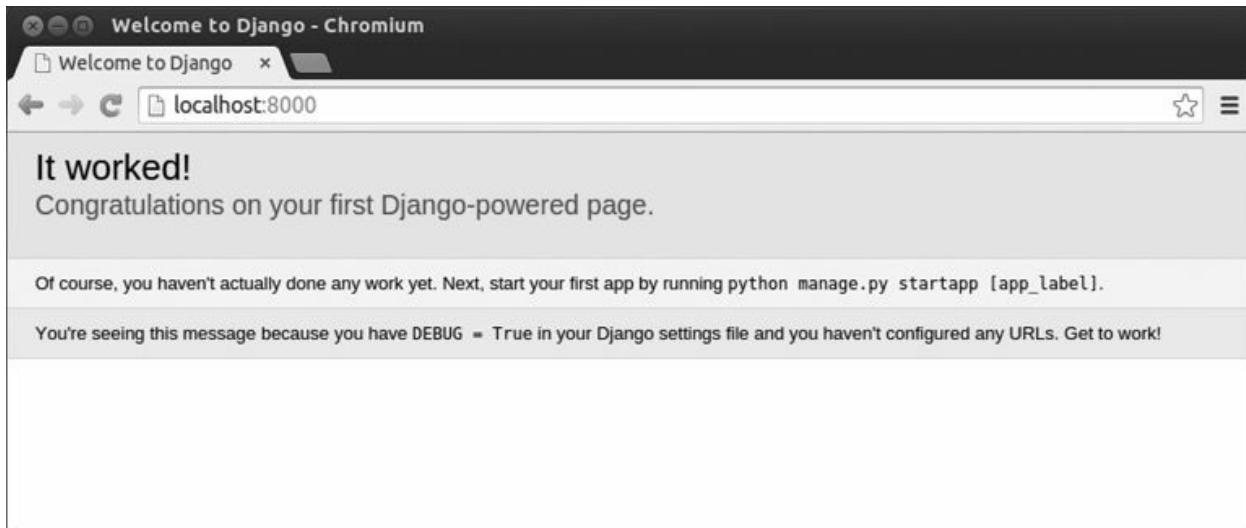



Figura 18.1 – Até agora, tudo está funcionando.

NOTA Se você receber a mensagem de erro *That port is already in use* (Essa porta já está em uso), diga a Django para usar uma porta diferente por meio do comando `python manage.py runserver 8001` e vá passando por números maiores até encontrar uma porta que esteja aberta.

FAÇA VOCÊ MESMO

18.1 – Novos projetos: Para ter uma ideia melhor do que Django faz, crie dois

projetos vazios e observe o que o framework criou. Crie uma nova pasta com um nome simples, por exemplo, *InstaBook* ou *FaceGram* (fora de seu diretório *learning_log*), navegue até essa pasta em um terminal e crie um ambiente virtual. Instale Django e execute o comando `django-admin.py startproject instabook .` (lembre-se de incluir o ponto no final do comando).

Observe os arquivos e as pastas criados por esse comando e compare-os com os de Learning Log. Faça isso algumas vezes até ter familiaridade com o que Django cria quando um novo projeto é iniciado. Então apague os diretórios dos projetos se quiser.

Iniciando uma aplicação

Um *projeto* Django é organizado na forma de um grupo de *aplicações* (apps) individuais que operam em conjunto para fazer o projeto funcionar como um todo. Por enquanto, criaremos apenas uma aplicação para fazer a maior parte do trabalho de nosso projeto. Adicionaremos outra aplicação para administrar contas de usuários no Capítulo 19.

Continue executando `runserver` na janela do terminal aberta anteriormente. Abra uma nova janela de terminal (ou aba) e navegue até o diretório que contém o arquivo *manage.py*. Ative o ambiente virtual e execute o comando `startapp`:

```
  learning_log$ source ll_env/bin/activate
  (ll_env)learning_log$ python manage.py startapp learning_logs
u (ll_env)learning_log$ ls
  db.sqlite3  learning_log  learning_logs  ll_env  manage.py
v (ll_env)learning_log$ ls learning_logs/
  admin.py  __init__.py  migrations  models.py  tests.py  views.py
```

O comando `startapp` *`nomeapp`* diz a Django para criar a infraestrutura necessária à construção de uma aplicação. Se você observar o diretório de projeto agora, verá uma nova pasta chamada *learning_logs* u. Abra essa pasta para ver o que Django criou v. Os arquivos mais importantes são: *models.py*, *admin.py* e *views.py*. Usaremos *models.py* para definir os dados que queremos administrar em nossa aplicação. Discutiremos *admin.py* e *views.py* um pouco mais adiante.

Definindo modelos

Vamos pensar em nossos dados por um instante. Cada usuário deverá criar vários assuntos em seu registro de aprendizado. Cada entrada criada estará associada a um assunto, e essas entradas serão exibidas como texto. Também será necessário armazenar o timestamp de cada entrada para que possamos mostrar aos usuários a data em que cada entrada foi criada.

Abra o arquivo *models.py* e observe o conteúdo existente:

models.py

```
from django.db import models

# Create your models here.
```

Um módulo chamado `models` foi importado para nós e somos convidados a criar nossos próprios modelos. Um modelo diz a Django como trabalhar com os dados que serão armazenados na aplicação. Do ponto de vista do código, um modelo é apenas uma classe; ele tem atributos e métodos, assim como todas as classes que discutimos. Eis o modelo para os assuntos que os usuários armazenarão:

```
  from django.db import models

  class Topic(models.Model):
      """Um assunto sobre o qual o usuário está aprendendo."""
u     text = models.CharField(max_length=200)
v     date_added = models.DateTimeField(auto_now_add=True)

w     def __str__(self):
          """Devolve uma representação em string do modelo."""
          return self.text
```

Criamos uma classe chamada `Topic`, que herda de `Model` – uma classe-pai incluída em Django, que define a funcionalidade básica de um modelo. A classe `Topic` tem apenas dois atributos: `text` e `date_added`.

O atributo `text` é um `CharField` – um dado composto de caracteres, isto é, um texto **u**. Usamos `CharField` quando queremos armazenar uma pequena quantidade de texto, por exemplo, um nome, um título ou uma cidade. Quando definimos um atributo `CharField`, devemos dizer a Django quanto espaço deve ser reservado no banco de dados. Nesse caso, especificamos

um `max_length` de 200 caracteres, que deverá ser suficiente para armazenar a maioria dos nomes de assuntos.

O atributo `date_added` é um `DateTimeField` – um dado que registrará uma data e uma hora **V**. Passamos o argumento `auto_now_add=True`, que diz a Django para definir esse atributo automaticamente com a data e hora atuais sempre que o usuário criar um novo assunto.

> **NOTA** Para ver os diferentes tipos de campos que você pode usar em um modelo, consulte o *Django Model Field Reference* (Referência aos campos do modelo de Django) em *https://docs.djangoproject.com/en/1.8/ref/models/fields/*. Você não precisará de todas as informações de imediato, mas elas serão extremamente úteis quando você desenvolver suas próprias aplicações.

Devemos dizer a Django qual atributo deve ser usado como default quando ele exibir informações sobre um assunto. O Django chama um método `__str__()` para exibir uma representação simples de um modelo. Nesse caso, escrevemos um método `__str__()` que devolve a string armazenada no atributo `text` **W**.

> **NOTA** Se você usa Python 2.7, deve chamar o método `__str__()` de `__unicode__()`. O corpo do método é idêntico.

Ativando os modelos

Para usar nossos modelos, devemos dizer a Django para incluir nossa aplicação no projeto como um todo. Abra *settings.py* (no diretório *learning_log/learning_log*) e você verá uma seção que informa a Django quais aplicações estão instaladas no projeto:

settings.py

```
--trecho omitido--
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
--trecho omitido--
```

É apenas uma tupla que informa a Django quais aplicações funcionam em conjunto para compor o projeto. Adicione sua aplicação nessa tupla modificando `INSTALLED_APPS` de modo que ela fique assim:

```
--trecho omitido--
INSTALLED_APPS = (
    --trecho omitido--
    'django.contrib.staticfiles',

    # Minhas aplicações
    'learning_logs',
)
--trecho omitido--
```

Agrupar aplicações em um projeto ajuda a manter o controle sobre elas à medida que o projeto crescer incluindo mais aplicações. Nesse caso, iniciamos uma seção chamada *Minhas aplicações*, que inclui apenas `learning_logs` por enquanto.

Em seguida, devemos dizer a Django para modificar o banco de dados para que ele possa armazenar informações relacionadas ao modelo `Topic`. A partir do terminal, execute o seguinte comando:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  0001_initial.py:
    - Create model Topic
(ll_env)learning_log$
```

O comando `makemigrations` diz a Django para descobrir como modificar o banco de dados para que ele possa armazenar os dados associados a qualquer novo modelo que definirmos. A saída, nesse caso, mostra que Django criou um arquivo de migração chamado *0001_initial.py*. Essa migração criará uma tabela para o modelo `Topic` no banco de dados.

Agora aplicaremos essa migração e faremos Django modificar o banco de dados para nós:

```
(ll_env)learning_log$ python manage.py migrate
--trecho omitido--
Running migrations:
```

```
   Rendering model states... DONE
u   Applying learning_logs.0001_initial... OK
```

A maior parte da saída desse comando é idêntica àquela obtida na primeira vez que executamos o comando `migrate`. A linha que devemos verificar aparece em u, na qual Django confirma que tudo funcionou bem (`OK`) quando a migração para `learning_logs` foi aplicada.

Sempre que quisermos modificar os dados administrados por Learning Log, executaremos estes três passos: modificaremos *models.py*, chamaremos `makemigrations` em `learning_logs` e diremos a Django para executar um `migrate` no projeto.

Site de administração de Django

Ao definir modelos para uma aplicação, o Django fará com que seja mais fácil para você trabalhar com seus modelos por meio do *site de administração* (admin site). São os administradores de um site que utilizam o site de administração, e não os usuários comuns. Nesta seção criaremos o site de administração e o usaremos para adicionar alguns assuntos por meio do modelo `Topic`.

Criando um superusuário

O Django permite criar um usuário com todos os privilégios disponíveis no site; esse usuário é conhecido como *superusuário*. Um *privilégio* controla as ações que um usuário pode executar. As configurações mais restritivas de privilégios permitem que um usuário apenas leia informações públicas do site. Usuários cadastrados normalmente têm privilégio para ler seus próprios dados privados e algumas informações selecionadas, disponíveis somente aos membros. Para administrar uma aplicação web de modo eficiente, o proprietário do site normalmente precisa ter acesso a todas as informações armazenadas no site. Um bom administrador é cuidadoso com as informações sensíveis de seus usuários, pois eles confiam bastante nas aplicações que acessam.

Para criar um superusuário em Django, execute o comando a seguir e responda aos prompts:

```
  (ll_env)learning_log$ python manage.py createsuperuser
u Username (leave blank to use 'ehmatthes'): ll_admin
v Email address:
w Password:
  Password (again):
  Superuser created successfully.
  (ll_env)learning_log$
```

Quando você executa o comando `createsuperuser`, o Django pede que você forneça um nome para o superusuário **u**. Nesse caso usamos *ll_admin*, mas você pode fornecer qualquer nome de usuário que quiser. Você pode especificar um endereço de email se desejar, ou pode simplesmente deixar esse campo em branco **v**. Será necessário digitar sua senha duas vezes **w**.

NOTA Algumas informações sensíveis podem ser ocultas dos administradores de um site. Por exemplo, o Django não armazena realmente a senha que você inserir; em vez disso, ele armazena uma string derivada da senha, conhecida como *hash*. Sempre que você fornecer sua senha, o Django calcula o hash de sua entrada e o compara com o hash armazenado. Se os dois hashes coincidirem, você será autenticado. Ao exigir que os hashes sejam iguais, caso um invasor obtenha acesso ao banco de dados de um site, ele poderá ler os hashes armazenados, mas não as senhas. Quando um site é configurado de modo apropriado, é quase impossível obter as senhas originais a partir dos hashes.

Registrando um modelo junto ao site de administração

O Django inclui alguns modelos no site de administração de modo automático, por exemplo, `User` e `Group`, mas os modelos que criamos devem ser registrados manualmente.

Quando iniciamos a aplicação `learning_logs`, o Django criou um arquivo chamado *admin.py* no mesmo diretório em que está *models.py*:

admin.py

```
from django.contrib import admin

# Register your models here.
```

Para registrar `Topic` junto ao site de administração, digite:

```
from django.contrib import admin
```

```
u from learning_logs.models import Topic

v admin.site.register(Topic)
```

Esse código importa o modelo que queremos registrar, `Topic` **u**, e então usa `admin.site.register()` **v** para dizer a Django que administre nosso modelo por meio do site de administração.

Agora utilize a conta do superusuário para acessar o site de administração. Vá para *http://localhost:8000/admin/*, forneça o nome e a senha do superusuário que você acabou de criar; você deverá ver uma tela como a que está na Figura 18.2. Essa página permite adicionar novos usuários e grupos, além de alterar aqueles que já existem. Também podemos trabalhar com dados relacionados ao modelo `Topic` que acabamos de definir.

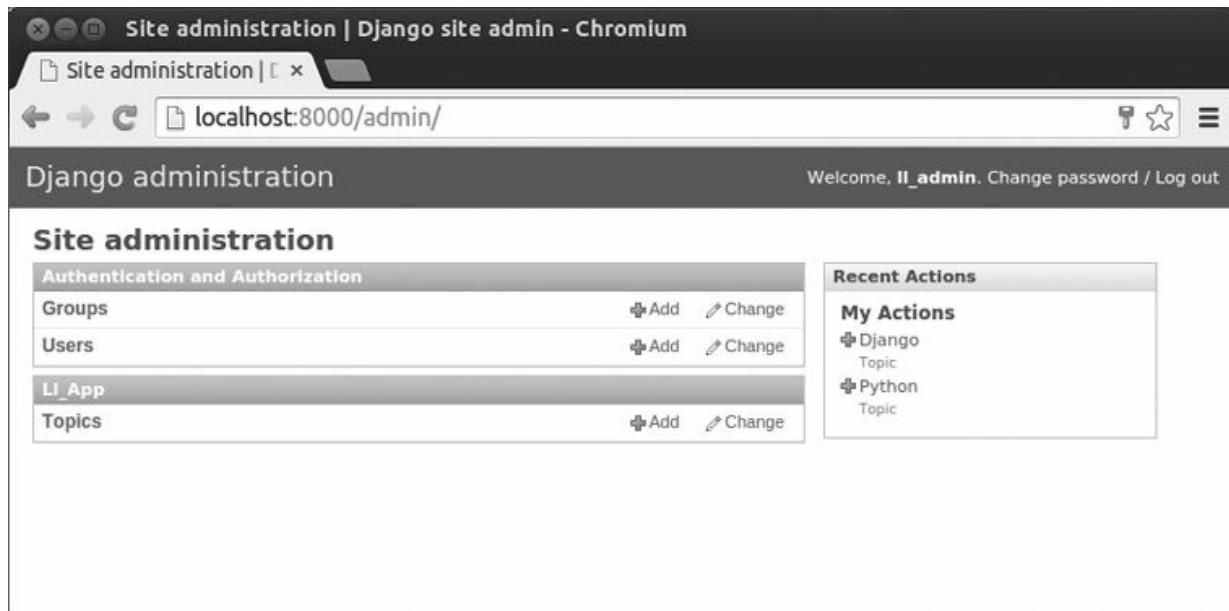



Figura 18.2 – O site de administração com `Topic` incluído.

NOTA Se você vir uma mensagem em seu navegador informando que a página web não está disponível, certifique-se de que o servidor Django continua executando em uma janela de terminal. Se não estiver, ative um ambiente virtual e execute o comando `python manage.py runserver` novamente.

Adicionando assuntos

Agora que `Topic` foi registrado no site de administração, vamos adicionar o

nosso primeiro assunto. Clique em **Topics** para acessar sua página que, em sua maior parte, estará vazia, pois ainda não temos nenhum assunto para administrar. Clique em **Add** (Adicionar) e você verá um formulário para adicionar um novo assunto. Insira **`Chess`** na primeira caixa e clique em **Save** (Salvar). Você será enviado de volta à página de administração de Topics e verá o assunto que acabou de criar.

Vamos criar um segundo assunto para que tenhamos mais dados para trabalhar. Clique em **Add** novamente e crie um segundo assunto: **`Rock Climbing`**. Quando clicar em **Save**, você será enviado de volta para a página principal de Topics novamente e verá tanto Chess (Xadrez) quando Rock Climbing (Escalada) listados.

Definindo o modelo Entry

Para registrar o que aprendemos sobre xadrez e escalada, precisamos definir um modelo para os tipos de entrada que os usuários podem criar em seus registros de aprendizado. Cada entrada deve estar associada a um assunto em particular. Esse relacionamento é chamado de *relacionamento de muitos para um* (many-to-one relationship), o que quer fizer que várias entradas podem estar associadas a um assunto.

Eis o código do modelo `Entry`:

models.py

```
  from django.db import models

  class Topic(models.Model):
      --trecho omitido--

u class Entry(models.Model):
      """Algo específico aprendido sobre um assunto."""
v     topic = models.ForeignKey(Topic)
w     text = models.TextField()
      date_added = models.DateTimeField(auto_now_add=True)

x     class Meta:
          verbose_name_plural = 'entries'
```

```
    def __str__(self):
        """Devolve uma representação em string do modelo."""
y       return self.text[:50] + "..."
```

A classe Entry herda da classe base Model de Django, assim como foi feito com Topic u. O primeiro atributo, topic, é uma instância de ForeignKey v. Uma *chave estrangeira* (foreign key) é um termo usado em banco de dados: é uma referência a outro registro do banco de dados. Esse é o código que associa cada entrada a um assunto específico. Cada assunto recebe uma chave, isto é, um ID, quando é criado. Quando Django precisa estabelecer uma conexão entre dois dados, ele usa a chave associada a cada informação. Utilizaremos essas conexões em breve para recuperar todas as entradas associadas a determinado assunto.

Em seguida, temos um atributo chamado text, que é uma instância de TextField w. Esse tipo de campo não precisa de um limite para o tamanho, pois não queremos restringir o tamanho das entradas individuais. O atributo date_added nos permite apresentar as entradas na ordem em que foram criadas e inserir um timestamp junto a cada entrada.

Em x aninhamos a classe Meta em nossa classe Entry. Meta armazena informações extras para administrar um modelo; nesse caso, ela nos permite definir um atributo especial que diz a Django para usar *Entries* quando precisar se referir a mais de uma entrada. (Sem isso, Django iria referenciar várias entradas como *Entrys*.) Por fim, o método __str__() diz a Django quais informações devem ser mostradas quando entradas individuais forem referenciadas. Como uma entrada pode ser um texto longo, dizemos a Django para mostrar apenas os primeiros 50 caracteres de text y. Também acrescentamos reticências para deixar claro que não estamos exibindo a entrada completa.

Migrando o modelo Entry

Como adicionamos um novo modelo, devemos migrar o banco de dados novamente. Esse processo se tornará bastante familiar: você modifica *models.py*, executa o comando python manage.py makemigrations *nome_app* e então executa python manage.py migrate.

Faça a migração do banco de dados e verifique a saída:

```
  (ll_env)learning_log$ python manage.py makemigrations learning_logs
  Migrations for 'learning_logs':
u   0002_entry.py:
      - Create model Entry
  (ll_env)learning_log$ python manage.py migrate
  Operations to perform:
    --trecho omitido--
v   Applying learning_logs.0002_entry... OK
```

Uma nova migração chamada *0002_entry.py* é gerada; ela diz a Django como o banco de dados deve ser modificado para armazenar informações relacionadas ao modelo `Entry` **u**. Quando executamos o comando `migrate`, vemos que Django aplicou essa migração e que tudo correu bem **v**.

Registrando Entry junto ao site de administração

Também precisamos registrar o modelo `Entry`. Eis a aparência do código em *admin.py* agora:

admin.py

```
from django.contrib import admin

from learning_logs.models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Acesse *http://localhost/admin/* novamente, e você deverá ver *Entries* listada em *learning_logs*. Clique no link **Add** (Adicionar) de Entries, ou clique em **Entries** (Entradas), e escolha **Add entry** (Adicionar entrada). Você deverá ver uma lista suspensa para selecionar o assunto para o qual você está criando uma entrada e uma caixa de texto para adicionar a entrada. Escolha **Chess** na lista suspensa e acrescente uma entrada. Eis a primeira entrada que eu criei:

> *The opening is the first part of the game, roughly the first ten moves or so. In the opening, it's a good idea to do three things—bring out your bishops and knights, try to control the center of the board, and castle your king.*

Of course, these are just guidelines. It will be important to learn when to follow these guidelines and when to disregard these suggestions.

[A abertura é a primeira parte do jogo; de modo geral, corresponde a algo em torno dos dez primeiros movimentos. Na abertura, é uma boa ideia fazer três coisas: avançar seus bispos e cavalos, tentar controlar o centro do tabuleiro e proteger seu rei com um roque.

É claro que essas são apenas orientações. Será importante saber quando você deve seguir essas orientações e quando deve deixar essas sugestões de lado.]

Quando clicar em **Save** (Salvar), você será levado de volta à página principal de administração para as entradas. Nesse ponto você verá a vantagem de usar `text[:50]` como a representação de string para cada entrada; é muito mais fácil trabalhar com várias entradas na interface de administração se você vir apenas a primeira parte de uma entrada, e não o texto todo.

Crie uma segunda entrada para Chess e uma para Rock Climbing para que tenhamos alguns dados iniciais. Eis uma segunda entrada para Chess:

In the opening phase of the game, it's important to bring out your bishops and knights. These pieces are powerful and maneuverable enough to play a significant role in the beginning moves of a game.

[Na fase de abertura do jogo, é importante avançar com seus bispos e cavalos. Essas peças são poderosas e podem se movimentar bastante a ponto de exercerem um papel significativo nos movimentos iniciais de um jogo.]

E aqui está a primeira entrada para Rock Climbing:

One of the most important concepts in climbing is to keep your weight on your feet as much as possible. There's a myth that climbers can hang all day on their arms. In reality, good climbers have practiced specific ways of keeping their weight over their feet whenever possible.

[Um dos conceitos mais importantes em escalada é manter o seu peso sobre seus pés o máximo possível. Existe um mito segundo o qual os alpinistas são capazes de ficar pendurados o dia todo usando os

braços. Na verdade, bons alpinistas treinam maneiras específicas de manter o seu peso sobre seus pés sempre que for possível.]

Essas três entradas nos darão algo com que trabalhar à medida que continuarmos o desenvolvimento do projeto Learning Log.

Shell de Django

Agora que inserimos alguns dados, podemos analisá-los por meio de programação em uma sessão interativa de terminal. Esse ambiente interativo é chamado de *shell* do Django, e é um ótimo ambiente para testar e resolver problemas de seu projeto. Eis um exemplo de uma sessão interativa de shell:

```
  (ll_env)learning_log$ python manage.py shell
u >>> from learning_logs.models import Topic
  >>> Topic.objects.all()
  [<Topic: Chess>, <Topic: Rock Climbing>]
```

O comando `python manage.py shell` (executado em um ambiente virtual ativo) inicia um interpretador Python que você pode usar para explorar os dados armazenados no banco de dados de seu projeto. Nesse caso importamos o modelo `Topic` do módulo `learning_logs.models` u. Então usamos o método `Topic.objects.all()` para obter todas as instâncias do modelo `Topic`; a lista devolvida se chama *queryset*.

Podemos percorrer um queryset do mesmo modo que o fazemos com uma lista. O ID atribuído a cada objeto que representa um assunto pode ser visto da seguinte maneira:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...   print(topic.id, topic)
...
1 Chess
2 Rock Climbing
```

Armazenamos o queryset em `topics` e então exibimos o atributo `id` e a representação em string de cada assunto. Podemos ver que Chess tem um ID igual a 1 e Rock Climbing tem ID igual a 2.

Se você souber qual é o ID de um objeto em particular, poderá acessar esse objeto e analisar qualquer atributo que ele tiver. Vamos observar os valores `text` e `date_added` de Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2015, 5, 28, 4, 39, 11, 989446, tzinfo=<UTC>)
```

Também podemos ver as entradas relacionadas a determinado assunto. Definimos anteriormente o atributo `topic` no modelo `Entry`. Esse atributo era um `ForeignKey`, isto é, uma conexão entre cada entrada e um assunto. O Django é capaz de usar essa conexão para obter todas as entradas relacionadas a determinado assunto, desta maneira:

```
u >>> t.entry_set.all()
  [<Entry: The opening is the first part of the game, roughly...>,
  <Entry: In the opening phase of the game, it's important t...>]
```

Para obter dados por meio de um relacionamento de chave estrangeira, utilize o nome do modelo relacionado com letras minúsculas, seguido de um underscore e da palavra `set` U. Por exemplo, suponha que você tenha os modelos `Pizza` e `Topping`, e `Topping` está relacionado a `Pizza` por meio de uma chave estrangeira. Se seu objeto se chamar `my_pizza`, representando uma única pizza, você poderá obter todos os ingredientes dessa pizza usando o código `my_pizza.topping_set.all()`.

Usaremos esse tipo de sintaxe quando começarmos a implementar as páginas que os usuários poderão pedir. O shell é bem conveniente para garantir que seu código recupere os dados que você quer. Se seu código funcionar conforme esperado no shell, você poderá esperar que ele funcione de modo apropriado nos arquivos de seu projeto. Se o código gerar erros ou não recuperar os dados esperados, será muito mais fácil resolver os problemas desse código no ambiente mais simples de shell que nos arquivos que geram as páginas web. Não vamos nos referir muito ao shell, mas você deve continuar a usá-lo para adquirir prática no uso da sintaxe de Django a fim de acessar os dados armazenados no projeto.

NOTA Sempre que modificar seus modelos, será necessário reiniciar o shell para ver os efeitos dessas alterações. Para sair de uma sessão de shell, tecle CTRL-D; no Windows, tecle CTRL-Z e depois ENTER.

FAÇA VOCÊ MESMO

18.2 – Entradas menores: No momento, o método `__str__()` no modelo `Entry` concatena reticências a todas as instâncias de `Entry` quando Django exibe uma entrada no site de administração ou no shell. Acrescente uma instrução `if` no método `__str__()` que adicione reticências somente se a entrada tiver mais de 50 caracteres. Utilize o site de administração para acrescentar uma entrada com menos de 50 caracteres e certifique-se de que ela não contenha reticências quando for visualizada.

18.3 – A API de Django: Quando escrever um código para acessar os dados de seu projeto, você estará escrevendo uma *query*. Dê uma olhada rápida na documentação sobre querying de seus dados em *https://docs.djangoproject.com/en/1.8/topics/db/queries/*. Muito do que você verá parecerá novidade, mas lhe será muito útil quando começar a trabalhar com seus próprios projetos.

18.4 – Pizzaria: Inicie um novo projeto chamado `pizzaria` com uma aplicação chamada `pizzas`. Defina um modelo `Pizza` com um campo chamado `name`, que armazenará nomes como `Hawaiian` e `Meat Lovers`. Defina um modelo chamado `Topping` com campos de nome `pizza` e `name`. O campo `pizza` deve ser uma chave estrangeira para `Pizza`, e `name` deve ser capaz de armazenar valores como `pineapple`, `Canadian bacon` e `sausage`.

Registre os dois modelos no site de administração e use esse site para fornecer alguns nomes de pizzas e de ingredientes. Utilize o shell para explorar os dados inseridos.

Criando páginas: a página inicial de Learning Log

Geralmente a criação de páginas web com Django é constituída de três etapas: definir os URLs, escrever as views e criar os templates. Em primeiro lugar, você deve definir padrões para os URLs. Um padrão de URL descreve o modo como o URL é organizado, e diz a Django o que ele deve procurar quando fizer a correspondência entre uma requisição do navegador e o URL de um site para que ele possa saber qual página deverá devolver.

Cada URL então é mapeado para uma *view* em particular – a função de

view obtém e processa os dados necessários a essa página. Essa função geralmente chama um *template*, que constrói uma página possível de ser lida por um navegador. Para ver como isso funciona, vamos criar a página inicial de Learning Log. Definiremos o URL da página inicial, escreveremos sua função de view e criaremos um template simples.

Como tudo que estamos fazendo é garantir que Learning Log funcione como deveria, deixaremos a página simples por enquanto. É divertido estilizar uma aplicação web funcional quando ela estiver completa; uma aplicação que tenha uma boa aparência, mas que não funcione bem não faz sentido. Por enquanto, a página inicial exibirá apenas um título e uma breve descrição.

Mapeando um URL

Os usuários solicitam páginas fornecendo URLs em um navegador e clicando em links, portanto precisamos decidir quais URLs são necessários em nosso projeto. O URL da página inicial é o primeiro: é o URL base que as pessoas usarão para acessar o projeto. No momento, o URL base, *http://localhost:8000/*, devolve o site default de Django, que nos permite saber se o projeto foi configurado de forma correta. Mudaremos isso mapeando o URL base para a página inicial de Learning Log.

Na pasta principal do projeto *learning_log*, abra o arquivo *urls.py*. Eis o código que você verá:

urls.py

```
u from django.conf.urls import include, url
  from django.contrib import admin

v urlpatterns = [
w     url(r'^admin/', include(admin.site.urls)),
  ]
```

As duas primeiras linhas importam as funções e módulos que administram os URLs do projeto e do site de administração u. O corpo do arquivo define a variável `urlpatterns` v. Nesse arquivo *urls.py*, que representa o projeto como um todo, a variável `urlpatterns` inclui os

conjuntos de URLs das aplicações do projeto. O código em w inclui o módulo `admin.site.urls`; ele define todos os URLs que podem ser requisitados a partir do site de administração.

Devemos incluir os URLs para `learning_logs`:

```
  from django.conf.urls import include, url
  from django.contrib import admin

  urlpatterns = [
      url(r'^admin/', include(admin.site.urls)),
u     url(r'', include('learning_logs.urls', namespace='learning_logs')),
  ]
```

Adicionamos uma linha para incluir o módulo `learning_logs.urls` em u. Essa linha inclui um argumento `namespace`, que nos permite distinguir os URLs de `learning_logs` de outros URLs que possam surgir no projeto, o que pode ser muito útil à medida que seu projeto começar a crescer.

O arquivo *urls.py* default está na pasta *learning_log*; agora precisamos criar um segundo arquivo *urls.py* na pasta *learning_logs*:

urls.py

```
u """Define padrões de URL para learning_logs."""

v from django.conf.urls import url

w from . import views

x urlpatterns = [
      # Página inicial
y     url(r'^$', views.index, name='index'),
  ]
```

Para deixar claro com qual *urls.py* estamos trabalhando, adicionamos uma docstring no início do arquivo u. Então importamos a função `url`, necessária ao mapear os URLs às views v. Também importamos o módulo `views` w; o ponto diz a Python para importar views do mesmo diretório em que está o módulo *urls.py* atual. A variável `urlpatterns` nesse módulo é uma lista de páginas individuais que podem ser solicitadas a partir da aplicação

`learning_logs` X.

O padrão de URL propriamente dito é uma chamada à função `url()`, que aceita três argumentos Y. O primeiro é uma expressão regular. Django procurará uma expressão regular em `urlpatterns` que corresponda à string do URL requisitado. Assim, uma expressão regular definirá o padrão que Django poderá procurar.

Vamos analisar a expressão regular `r'^$'`. O `r` diz a Python para interpretar a string que se segue como uma string pura, e as aspas dizem em que ponto a expressão regular começa e em que ponto ela termina. O acento circunflexo (^) diz a Python para localizar o início da string, e o sinal de cifrão diz para procurar o final dela. Como um todo, essa expressão diz a Python para procurar um URL sem nada entre o início e o fim do URL. Python ignora o URL-base do projeto (*http://localhost:8000/*), portanto uma expressão regular vazia corresponde ao URL-base. Qualquer outro URL não corresponderá a essa expressão, e Django devolverá uma página de erro se o URL requisitado não corresponder a nenhum padrão de URL existente.

O segundo argumento de `url()` em Y especifica qual função de view deve ser chamada. Quando um URL solicitado corresponder à expressão regular, Django chamará `views.index` (escreveremos essa função de view na próxima seção). O terceiro argumento fornece o nome `index` para esse padrão de URL para que possamos referenciá-lo em outras seções do código. Sempre que quisermos disponibilizar um link para a página inicial, usaremos esse nome em vez de escrever um URL.

NOTA As expressões regulares, com frequência chamadas de *regexes*, são usadas em quase todas as linguagens de programação. São extremamente úteis, mas exigem um pouco de prática para se acostumar com elas. Se você não compreendeu tudo, não se preocupe; você verá muitos exemplos à medida que trabalhar neste projeto.

Escrevendo uma view

Uma função de view recebe informações de uma requisição, prepara os dados necessários para gerar uma página e então envia os dados de volta ao navegador, geralmente usando um template que define a aparência da

página.

O arquivo *views.py* em *learning_logs* foi gerado automaticamente quando executamos o comando `python manage.py startapp`. Eis o código que está em *views.py* agora:

views.py

```
from django.shortcuts import render

# Create your views here.
```

No momento, esse arquivo simplesmente importa a função `render()`, que renderiza a resposta de acordo com os dados fornecidos pelas views. O código a seguir mostra como a view para a página inicial deve ser escrita:

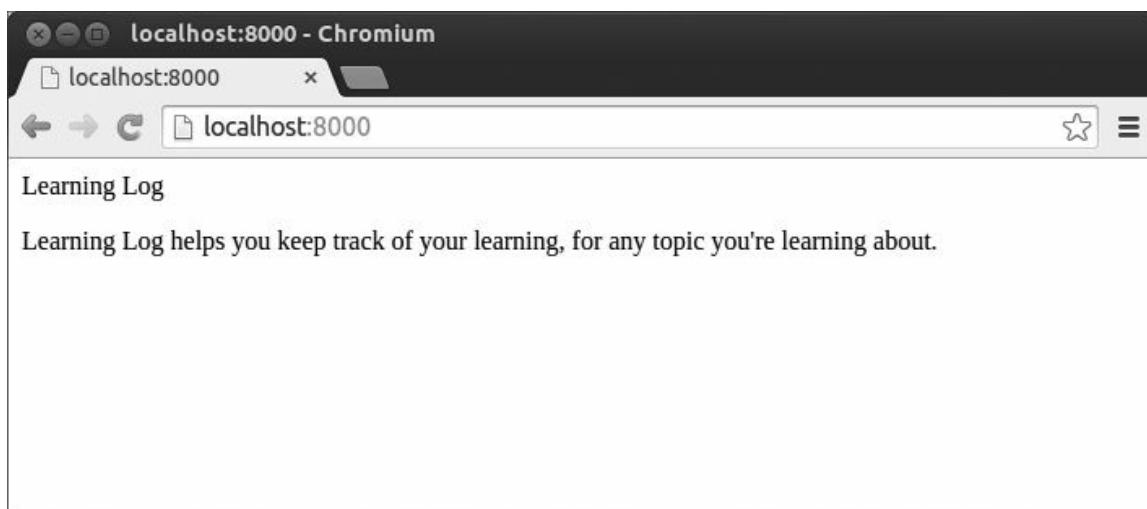
```
from django.shortcuts import render

def index(request):
    """A página inicial de Learning Log"""
    return render(request, 'learning_logs/index.html')
```

Quando uma requisição de URL corresponder ao padrão que acabamos de definir, o Django procurará uma função chamada `index()` no arquivo *views.py*. Então o objeto `request` será passado por Django para essa função de view. Nesse caso, não há necessidade de processar nenhum dado para a página, portanto o único código da função é uma chamada a `render()`. A função `render()` utiliza dois argumentos – o objeto `request` original e um template que pode ser usado para construir a página. Vamos criar esse template.

Escrevendo um template

Um template define a estrutura de uma página web. Ele define a aparência da página, e Django preencherá os dados relevantes sempre que a página for solicitada. Um template permite acessar qualquer dado oferecido pela view. Como nossa view para a página inicial não forneceu nenhum dado, esse template será bem simples.

Na pasta *learning_logs*, crie uma nova pasta chamada *templates*. Nessa pasta, crie outra pasta chamada *learning_logs*. Pode parecer um pouco

redundante (tempos uma pasta de nome *learning_logs* em uma pasta chamada *templates* que está em uma pasta chamada *learning_logs*), mas isso define uma estrutura que Django é capaz de interpretar sem que haja ambiguidades, mesmo no contexto de um projeto grande, com várias aplicações individuais. Na pasta *learning_logs* interna, crie um novo arquivo de nome *index.html*. Escreva o seguinte nesse arquivo:

index.html

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic
you're
learning about.</p>
```

Esse é um arquivo bem simples. Caso você não tenha familiaridade com HTML, as tags `<p></p>` representam parágrafos. A tag `<p>` inicia um parágrafo, enquanto a tag `</p>` o encerra. Temos dois parágrafos: o primeiro atua como um título e o segundo descreve o que os usuários de Learning Log podem fazer.

Agora, quando requisitarmos o URL base do projeto, *http://localhost:8000/*, veremos a página que acabamos de construir, em vez de ver a página default de Django. O Django lerá o URL requisitado, e esse URL corresponderá ao padrão `r'^$'`; então a função `views.index()` será chamada e a página será renderizada com o template que está em *index.html*. A página resultante pode ser vista na Figura 18.3.

Figura 18.3 – A página inicial de Learning Log.

Embora possa parecer um processo complicado para criar uma página, essa separação entre URLs, views e templates, na verdade, funciona bem. Ela permite pensar em cada aspecto de um projeto separadamente; em projetos maiores, isso possibilita que os indivíduos se concentrem nas áreas em que são mais capacitados. Por exemplo, um especialista em banco de dados poderá se concentrar nos modelos, um programador poderá ter como foco o código das views e um web designer poderá se concentrar nos templates.

FAÇA VOCÊ MESMO

18.5 – Planejamento de refeições: Considere uma aplicação que ajude as pessoas a planejar suas refeições ao longo da semana. Crie uma nova pasta chamada *meal_planner* e inicie um novo projeto Django nessa pasta. Então crie uma nova aplicação chamada `meal_plans`. Crie uma página inicial simples para esse projeto.

18.6 – Página inicial da pizzaria: Acrescente uma página inicial para o projeto *Pizzaria* que você começou a implementar no Exercício 18.4 (página 523).

Construindo páginas adicionais

Agora que estabelecemos uma rotina para construir uma página, podemos começar a desenvolver o projeto Learning Log. Criaremos duas páginas para exibição de dados: uma página que lista todos os assuntos e outra que mostra todas as entradas associadas a um assunto em particular. Para cada uma dessas páginas, especificaremos um padrão de URL, escreveremos uma função de view e criaremos um template. Porém, antes disso, vamos criar um template-base do qual todos os templates do projeto poderão herdar.

Herança de templates

Quando criar um site, quase sempre você precisará de alguns elementos que se repetirão em todas as páginas. Em vez de escrever esses elementos diretamente em cada página, você poderá criar um template base que

contenha os elementos repetidos e então fazer cada página herdar desse template. Essa abordagem permite que o enfoque seja dado no desenvolvimento dos aspectos exclusivos de cada página e facilita bastante alterar a aparência do projeto como um todo.

Template-pai

Começaremos criando um template chamado *base.html* no mesmo diretório em que está *index.html*. Esse arquivo conterá elementos comuns a todas as páginas; todos os demais templates herdarão de *base.html*. O único elemento que queremos repetir em todas as páginas no momento é o título na parte superior. Como incluiremos esse template em todas as páginas, vamos fazer com que o título seja um link para a página inicial:

base.html

```
  <p>
u   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
  </p>

v {% block content %}{% endblock content %}
```

A primeira parte desse arquivo cria um parágrafo contendo o nome do projeto, que também atua como um link para a página inicial. Para gerar um link, usamos uma *tag de template*, representada por chaves e os sinais de porcentagem `{% %}`. Uma tag de template é uma porção de código que gera informações a serem exibidas em uma página. Nesse exemplo, a tag de template `{% url 'learning_logs:index' %}` gera um URL que corresponde ao padrão de URL definido em *learning_logs/urls.py* cujo nome é `'index'` u. Nesse caso `learning_logs` é o *namespace* e `index` é um padrão de URL de nome único nesse namespace.

Em uma página HTML simples, um link é cercado pela tag de *âncora*:

```
<a href="url_do_link">texto do link</a>
```

Fazer a tag de template gerar o URL para nós facilita bastante para manter nossos links atualizados. Para mudar um URL em nosso projeto, basta alterar o padrão de URL em *urls.py* e Django inserirá automaticamente o URL atualizado na próxima vez que a página for

solicitada. Toda página de nosso projeto herdará de *base.html*, portanto, a partir de agora, toda página terá um link para a página inicial.

Em **v** inserimos um par de tags `block`. Esse bloco, que se chama `content`, é um placeholder; o template-filho definirá o tipo de informação que deverá ser inserido nesse bloco.

Um template-filho não precisa definir todos os blocos de seu pai, portanto você pode reservar espaço nos templates-pai para quantos blocos quiser, e o template-filho usará apenas quantos forem necessários.

> **NOTA** Em código Python, quase sempre indentamos com quatro espaços. Os arquivos de template tendem a ter mais níveis para aninhar que os arquivos Python, portanto é comum usar apenas dois espaços para cada nível de indentação.

Template-filho

Agora precisamos reescrever *index.html* para que herde de *base.html*. Eis o código de *index.html*:

index.html

```
u {% extends "learning_logs/base.html" %}

v {% block content %}
    <p>Learning Log helps you keep track of your learning, for any topic
  you're
    learning about.</p>
w {% endblock content %}
```

Se você comparar esse código com o código original de *index.html*, poderá ver que substituímos o título Learning Log pelo código para herdar de um template-pai **u**. Um template-filho deve ter uma tag `{% extends %}` na primeira linha para dizer a Django de qual template-pai ele deve herdar. O arquivo *base.html* faz parte de `learning_logs`, portanto incluímos *learning_logs* no path do template-pai. Essa linha extrai tudo que está contido no template *base.html* e permite que *index.html* defina o que deve ser colocado no espaço reservado pelo bloco `content`.

Definimos o bloco de conteúdo em **v** inserindo uma tag `{% block %}` com o nome `content`. Tudo que não herdamos do template-pai será colocado em

um bloco `content`. Nesse caso, é o parágrafo que descreve o projeto Learning Log. Em W informamos que acabamos de definir o conteúdo usando uma tag `{% endblock content %}`.

Você deve estar começando a perceber as vantagens da herança de templates: em um template-filho, só precisamos incluir o conteúdo que é exclusivo dessa página. Isso não só simplifica cada template como também facilita bastante modificar o site. Para alterar um elemento comum a várias páginas, basta modificar o elemento no template-pai. Suas alterações serão então propagadas para todas as páginas que herdam desse template. Em um projeto que inclua dezenas ou centenas de páginas, essa estrutura pode fazer com que seja muito mais fácil e rápido melhorar o seu site.

NOTA Em um projeto grande, é comum ter um template-pai chamado *base.html* para todo o site e templates-pai para cada seção principal do site. Todos os templates de seção herdam de *base.html*, e toda página do site herda de um template de seção. Desse modo, podemos modificar facilmente a aparência do site como um todo, de qualquer seção do site ou de uma página em particular. Essa configuração oferece uma maneira bem eficiente de trabalhar e incentiva você a atualizar constantemente o seu site com o passar do tempo.

Página de assuntos

Agora que temos uma abordagem eficiente para construir páginas, podemos nos concentrar em nossas duas próximas páginas: a página geral de assuntos e a página para exibir as entradas associadas a um único assunto. A página de assuntos mostrará todos os tópicos criados pelos usuários, e é a primeira página que envolverá manipulação de dados.

Padrão de URL para os assuntos

Inicialmente definiremos o URL para a página de assuntos. É comum escolher um fragmento simples de URL que reflita o tipo de informação apresentado na página. Utilizaremos a palavra *topics*, portanto o URL *http://localhost:8000/topics/* devolverá essa página. Eis o modo como modificamos *learning_logs/urls.py*:

urls.py

```
"""Define padrões de URL para learning_logs."""
```

```
  --trecho omitido--
  urlpatterns = [
      # Página inicial
      url(r'^$', views.index, name='index'),

      # Mostra todos os assuntos
u     url(r'^topics/$', views.topics, name='topics'),
  ]
```

Simplesmente adicionamos `topics/` no argumento da expressão regular usada para o URL da página inicial u. Quando Django analisar um URL solicitado, esse padrão corresponderá a qualquer URL que tenha o URL base seguido de *topics*. Podemos incluir ou omitir uma barra para a frente no final, mas não pode haver mais nada depois da palavra *topics*; do contrário, não haverá correspondência com o padrão. Qualquer requisição com um URL que corresponda a esse padrão será então passada para a função `topics()` em *views.py*.

View de assuntos

A função `topics()` precisa obter alguns dados do banco de dados e enviá-los ao template. A seguir está o código que devemos acrescentar em *views.py*:

views.py

```
  from django.shortcuts import render

u from .models import Topic

  def index(request):
      --trecho omitido--

v def topics(request):
      """Mostra todos os assuntos."""
w     topics = Topic.objects.order_by('date_added')
x     context = {'topics': topics}
y     return render(request, 'learning_logs/topics.html', context)
```

Inicialmente importamos o modelo associado aos dados de que

precisamos u. A função `topics()` exige um parâmetro: o objeto `request` que Django recebeu do servidor v. Em w consultamos o banco de dados pedindo os objetos `Topic`, ordenados de acordo com o atributo `date_added`. Armazenamos o queryset resultante em `topics`.

Em x definimos um contexto que será enviado ao template. Um *contexto* é um dicionário em que as chaves são os nomes que usaremos no template para acessar os dados e os valores são os dados que devemos enviar ao template. Nesse caso, há apenas um par chave-valor, que contém o conjunto de assuntos a ser exibido na página. Ao construir uma página que use dados, passamos a variável `context` para `render()`, além do objeto `request` e o path do template y.

Template para assuntos

O template para a página de assuntos recebe o dicionário `context` para que os dados fornecidos por `topics()` possam ser usados. Crie um arquivo chamado *topics.html* no mesmo diretório em que está *index.html*. Eis o modo como podemos exibir os assuntos no template:

topics.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

    <p>Topics</p>

u   <ul>
v     {% for topic in topics %}
w       <li>{{ topic }}</li>
x     {% empty %}
        <li>No topics have been added yet.</li>
y     {% endfor %}
z   </ul>

  {% endblock content %}
```

Começamos usando a tag `{% extends %}` para herdar de *base.html*, assim como fez o template de índice, e depois iniciamos um bloco `content`. O

corpo dessa página contém uma lista com marcadores contendo os assuntos fornecidos. Em HTML padrão, uma lista com marcadores é chamada de *lista não ordenada* e é representada pelas tags `<ul></ul>`. Iniciamos a lista de assuntos em u.

Em v temos outra tag de template equivalente a um laço `for`, que percorre a lista `topics` do dicionário `context`. O código usado nos templates difere de Python em alguns aspectos importantes. Python utiliza indentação para indicar quais linhas de uma instrução `for` fazem parte de um laço. Em um template, todo laço `for` deve ter uma tag `{% endfor %}` explícita para indicar em que ponto o laço termina. Desse modo, em um template, você verá laços escritos assim:

```
{% for item in lista %}
    faz algo com cada item
{% endfor %}
```

No laço, queremos transformar cada assunto em um item da lista com marcadores. Para exibir uma variável em um template, coloque o nome dela entre chaves duplas. O código `{{ topic }}` em w será substituído pelo valor de `topic` a cada passagem pelo laço. As chaves não aparecerão na página; elas simplesmente informam a Django que estamos usando uma variável de template. A tag HTML `<li></li>` representa um item de uma lista. Tudo que estiver entre as tags, em um par de tags `<ul></ul>`, aparecerá como um item marcado da lista.

Em x usamos a tag de template `{% empty %}`, que diz a Django o que deve ser feito se não houver nenhum item na lista. Nesse caso, exibimos uma mensagem informando ao usuário que, por enquanto, nenhum assunto foi adicionado. As duas últimas linhas encerram o laço `for` y e a lista com marcadores z.

Agora devemos modificar o template-base para que inclua um link para a página de assuntos:

base.html

```
  <p>
u   <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
v   <a href="{% url 'learning_logs:topics' %}">Topics</a>
```

```
</p>

{% block content %}{% endblock content %}
```

Acrescentamos um traço depois do link para a página inicial **u** e, em seguida, adicionamos um link para a página de assuntos, usando a tag de template para URL novamente **v**. Essa linha diz a Django para gerar um link que corresponda ao padrão de URL cujo nome é `'topics'` em *learning_logs/urls.py*.

Agora, quando atualizar a página inicial em seu navegador, você verá um link *Topics*. Ao clicar no link, você verá uma página semelhante àquela mostrada na Figura 18.4.

Figura 18.4 – A página de assuntos.

Páginas de assuntos individuais

A seguir, precisamos criar uma página para enfocar um único assunto, mostrando o nome desse assunto e todas as entradas associadas a ele. Novamente, definiremos um novo padrão de URL, escreveremos uma view e criaremos um template. Também modificaremos a página de assuntos para que cada item da lista com marcadores tenha um link para a sua página de assunto correspondente.

Padrão de URL para um assunto

O padrão de URL para a página de um assunto é um pouco diferente dos

demais padrões que vimos até agora porque ele usará o atributo `id` do assunto a fim de informar qual é o assunto solicitado. Por exemplo, se o usuário quiser ver a página de detalhes do assunto Chess, cujo `id` é 1, o URL será *http://localhost:8000/topics/1/*. A seguir vemos um padrão para corresponder a esse URL; esse padrão será incluído em *learning_logs/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    --trecho omitido--
    # Página de detalhes para um único assunto
    url(r'^topics/(?P<topic_id>\d+)/$', views.topic, name='topic'),
]
```

Vamos analisar a expressão regular nesse padrão de URL: `r'^topics/(?P<topic_id>\d+)/$'`. O `r` diz a Django para interpretar a string como uma string pura, e a expressão está entre aspas. A segunda parte da expressão, `/(?P<topic_id>\d+)/`, faz a correspondência de um inteiro entre duas barras para a frente e armazena esse valor em um argumento chamado `topic_id`. Os parênteses em torno dessa parte da expressão capturam o valor armazenado no URL; a parte `?P<topic_id>` armazena o valor correspondente em `topic_id` e a expressão `\d+` faz a correspondência de qualquer quantidade de dígitos que apareçam entre barras.

Quando Django encontrar um URL que corresponda a esse padrão, ele chamará a função de view `topic()` com o valor armazenado em `topic_id` como argumento. Usaremos o valor de `topic_id` para obter o assunto correto na função.

View de um assunto

A função `topic()` precisa obter o assunto e todas as entradas associadas a ele presentes no banco de dados, como vemos a seguir:

views.py

```
  --trecho omitido--
u def topic(request, topic_id):
```

```
      """Mostra um único assunto e todas as suas entradas."""
v     topic = Topic.objects.get(id=topic_id)
w     entries = topic.entry_set.order_by('-date_added')
x     context = {'topic': topic, 'entries': entries}
y     return render(request, 'learning_logs/topic.html', context)
```

Essa é a primeira função de view que exige um parâmetro que não seja o objeto `request`. A função aceita o valor capturado pela expressão `(?P<topic_id>\d+)` e o armazena em `topic_id` u. Em v usamos `get()` para obter o assunto, assim como fizemos no shell de Django. Em w recuperamos as entradas associadas a esse assunto e as ordenamos de acordo com `date_added`: o sinal de menos na frente de `date_added` ordena os resultados em ordem inversa, o que fará as entradas mais recentes serem exibidas antes. Armazenamos o assunto e as entradas no dicionário de contexto x e enviamos `context` para o template *topic.html* y.

NOTA Os códigos em v e em w são chamados de *queries*, pois fazem queries no banco de dados em busca de informações específicas. Ao escrever queries como essas em seus próprios projetos, será bem conveniente testá-las antes no shell de Django. Você terá um feedback muito mais rápido no shell do que teria se escrevesse uma view e um template e então conferisse os resultados em um navegador.

Template para um assunto

O template deve exibir o nome do assunto e as entradas. Também precisamos informar o usuário caso ainda não haja nenhuma entrada fornecida para esse assunto:

topic.html

```
  {% extends 'learning_logs/base.html' %}

  {% block content %}

u   <p>Topic: {{ topic }}</p>

    <p>Entries:</p>
v   <ul>
w   {% for entry in entries %}
      <li>
```

```
x       <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
y       <p>{{ entry.text|linebreaks }}</p>
      </li>
z   {% empty %}
      <li>
        There are no entries for this topic yet.
      </li>
    {% endfor %}
    </ul>

  {% endblock content %}
```

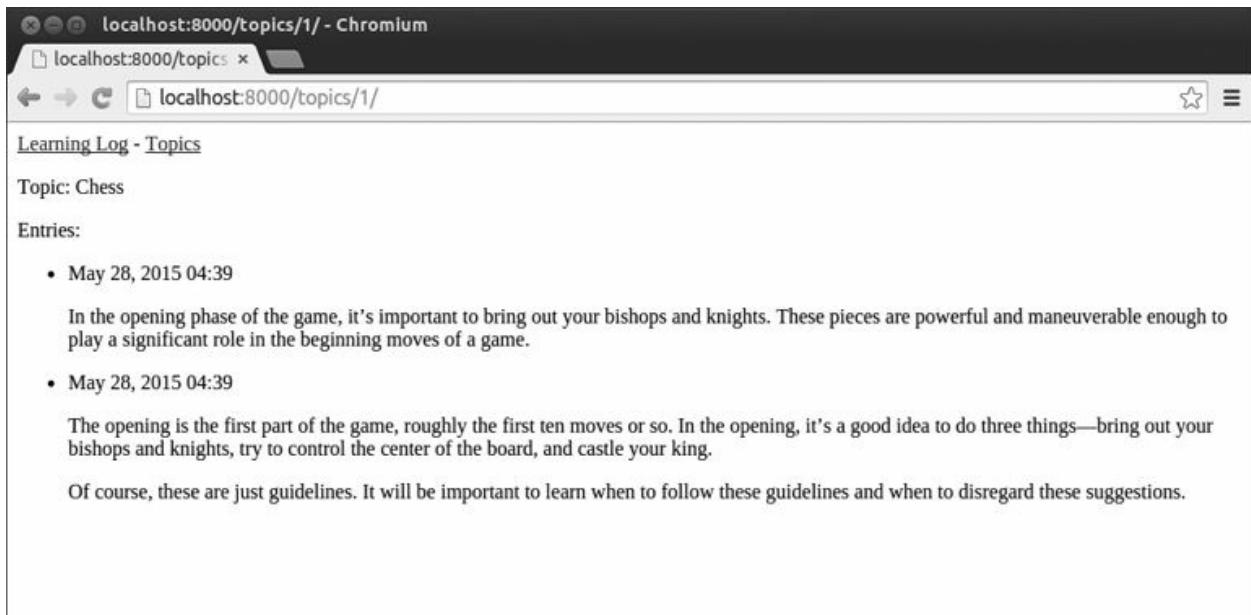
Estendemos *base.html* como fizemos para todas as páginas do projeto. Em seguida, mostramos o assunto exibido no momento u, que está armazenado na variável de template `{{ topic }}`. A variável `topic` está disponível porque está incluída no dicionário `context`. Então iniciamos uma lista com marcadores para mostrar cada uma das entradas v e as percorremos em um laço, como fizemos antes com os assuntos w.

Cada item da lista apresentará duas informações: o timestamp e o texto completo de cada entrada. Para o timestamp x, exibimos o valor do atributo `date_added`. Nos templates de Django, uma linha vertical (`|`) representa um *filtro* de template – uma função que modifica o valor de uma variável de template. O filtro `date:'M d, Y H:i'` exibe timestamps no formato *January 1, 2015 23:00*. A próxima linha exibe o valor completo de `text`, e não apenas os 50 primeiros caracteres de `entry`. O filtro `linebreaks` y garante que entradas com texto longo incluam quebras de linha em um formato compreensível pelos navegadores, em vez de mostrar um bloco de texto contínuo. Em z usamos a tag de template `{% empty %}` para exibir uma mensagem informando o usuário que nenhuma entrada foi fornecida.

Links a partir da página de assuntos

Antes de olhar a página de um assunto em um navegador, precisamos modificar o template de assuntos de modo que cada assunto tenha um link para a página apropriada. Eis a mudança feita em *topics.html*:

topics.html

```
  --trecho omitido--
```

```
    {% for topic in topics %}
      <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}
</a>
      </li>
    {% empty %}
--trecho omitido--
```

Usamos a tag de template para URL a fim de gerar o link apropriado, de acordo com o padrão de URL em `learning_logs` cujo nome é `'topic'`. Esse padrão de URL exige um argumento `topic_id`, portanto acrescentamos o atributo `topic.id` à tag de template para URL. Agora cada assunto da lista é um link para uma página de assunto, por exemplo, *http://localhost:8000/topics/1/*.

Se a página de assuntos for atualizada e você clicar em um assunto, uma página semelhante àquela mostrada na Figura 18.5 deverá ser apresentada.

Figura 18.5 – A página de detalhes para um único assunto, mostrando todas as entradas associadas a ele.

FAÇA VOCÊ MESMO

18.7 – Documentação de templates: Dê uma olhada rápida na documentação de templates de Django em *https://docs.djangoproject.com/en/1.8/ref/templates/*.

Você pode consultá-la novamente quando estiver trabalhando com seus próprios projetos.

18.8 – Páginas da pizzaria: Acrescente uma página ao projeto *Pizzaria* do Exercício 18.6 (página 529) que mostre os nomes das pizzas disponíveis. Então crie um link do nome de cada pizza para uma página que mostre os seus ingredientes. Lembre-se de usar herança de templates para construir suas páginas de modo eficiente.

Resumo

Neste capítulo começamos a ver como criar aplicações web usando o framework Django. Escrevemos uma breve especificação de projeto, instalamos o Django em um ambiente virtual, aprendemos a configurar um projeto e verificamos se o projeto foi criado corretamente. Aprendemos a configurar uma aplicação e definimos modelos para representar os dados de nossa aplicação. Conhecemos os bancos de dados e como Django ajuda a migrar o seu banco de dados depois que você fizer uma alteração em seus modelos. Aprendemos a criar um superusuário para o site de administração e usamos esse site para fornecer alguns dados iniciais.

Também exploramos o shell de Django, que permite trabalhar com os dados de seu projeto em uma sessão de terminal. Aprendemos a definir URLs, criar funções de view e escrever templates para criar as páginas de seu site. Por fim, usamos herança de template para simplificar a estrutura dos templates individuais e facilitar a modificação do site à medida que o projeto evoluir.

No Capítulo 19 criaremos páginas intuitivas e amigáveis que permitem aos usuários acrescentar novos assuntos e entradas e editar entradas existentes sem acessar o site de administração. Também adicionaremos um sistema de cadastro de usuários que permitirá que eles criem uma conta e façam seus próprios registros de aprendizado. Este é o coração de uma aplicação web: a capacidade de criar algo com que vários usuários possam interagir.

19

CONTAS DE USUÁRIO

No coração de uma aplicação web está a capacidade de qualquer usuário, em qualquer lugar do mundo, registrar uma conta junto à sua aplicação e começar a usá-la. Neste capítulo construiremos formulários para que os usuários possam adicionar seus próprios assuntos e entradas, além de editar as entradas existentes. Também veremos como Django se protege contra ataques comuns em páginas baseadas em formulário para que você não precise gastar muito tempo pensando na segurança de suas aplicações.

Assim, implementaremos um sistema de autenticação de usuários. Construiremos uma página de cadastro para os usuários criarem contas e então restringiremos o acesso a determinadas páginas apenas aos usuários logados. Em seguida modificaremos algumas das funções de view para que os usuários possam ver somente os seus próprios dados. Você aprenderá a manter os dados de seus usuários seguros e protegidos.

Permitindo que os usuários forneçam dados

Antes de desenvolver um sistema de autenticação para criar contas, vamos acrescentar algumas páginas que permitam aos usuários fornecer seus próprios dados. Daremos a eles a capacidade de adicionar um novo assunto ou uma nova entrada e editar as entradas anteriores.

No momento, apenas um superusuário pode inserir dados por meio do site de administração. Não queremos que os usuários interajam com o site de administração; desse modo, usaremos as ferramentas de construção de formulários do Django para criar páginas que permitam aos usuários fornecer dados.

Adicionando novos assuntos

Vamos começar dando aos usuários a capacidade de adicionar um novo assunto. Acrescentar uma página baseada em formulário é muito semelhante a adicionar as páginas que já criamos: definimos um URL, escrevemos uma função de view e criamos um template. A principal diferença está na adição de um novo módulo, chamado *forms.py*, que conterá os formulários.

ModelForm para assuntos

Qualquer página que permita a um usuário fornecer e submeter informações em uma página web é um *formulário*, mesmo que não se pareça com um. Quando os usuários fornecem informações, precisamos *validar* se as informações inseridas são do tipo correto, e não dados maliciosos, como um código para interromper o nosso servidor. Então devemos processar e salvar as informações válidas em um lugar apropriado no banco de dados. O Django automatiza boa parte dessas tarefas.

A maneira mais simples de construir um formulário em Django é usar um *ModelForm*, que utiliza as informações dos modelos que definimos no Capítulo 18 para criar automaticamente um formulário. Crie o seu primeiro formulário no arquivo *forms.py*, que deve estar no mesmo diretório em que está *models.py*:

forms.py

```
  from django import forms

  from .models import Topic

u class TopicForm(forms.ModelForm):
      class Meta:
v         model = Topic
w         fields = ['text']
x         labels = {'text': ''}
```

Inicialmente importamos o módulo `forms` e o modelo `Topic` com o qual trabalharemos. Em u definimos uma classe chamada `TopicForm` que herda de `forms.ModelForm`.

A versão mais simples de um `ModelForm` é constituída de uma classe `Meta`

aninhada que diz a Django em qual modelo o formulário deve se basear e quais campos devem ser incluídos nesse formulário. Em `v` criamos um formulário a partir do modelo `Topic` e incluímos apenas o campo `text` `w`. O código em `x` diz a Django para não gerar um rótulo para o campo `text`.

URL new_topic

O URL para uma nova página deve ser conciso e descritivo; assim, quando o usuário quiser adicionar um novo assunto, ele será enviado para *http://localhost:8000/new_topic/*. Eis o padrão de URL para a página `new_topic`, que acrescentaremos em *learning_logs/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    --trecho omitido--
    # Página para adicionar um novo assunto
    url(r'^new_topic/$', views.new_topic, name='new_topic'),
]
```

Esse padrão de URL enviará solicitações para a função de view `new_topic()`, que escreveremos em seguida.

Função de view new_topic()

A função `new_topic()` deve tratar duas situações diferentes: requisições iniciais para a página `new_topic` (caso em que um formulário em branco deverá ser mostrado) e o processamento de qualquer dado submetido no formulário. Então a função deverá redirecionar o usuário de volta à página `topics`:

views.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse

from .models import Topic
from .forms import TopicForm
```

```
  --trecho omitido--
  def new_topic(request):
      """Adiciona um novo assunto."""
u     if request.method != 'POST':
          # Nenhum dado submetido; cria um formulário em branco
v         form = TopicForm()
      else:
          # Dados de POST submetidos; processa os dados
w         form = TopicForm(request.POST)
x         if form.is_valid():
y             form.save()
z             return
HttpResponseRedirect(reverse('learning_logs:topics'))

{     context = {'form': form}
      return render(request, 'learning_logs/new_topic.html', context)
```

Importamos a classe `HttpResponseRedirect`, que usaremos para redirecionar o leitor de volta à página `topics`, depois que ele tiver submetido o seu assunto. A função `reverse()` determina o URL a partir de um padrão de URL nomeado, o que quer dizer que Django gerará o URL quando a página for solicitada. Também importamos `TopicForm`, que é o formulário que acabamos de criar.

Requisições GET e POST

Os dois principais tipos de requisição que você usará ao criar aplicações web são as requisições GET e POST. Usamos requisições *GET* para páginas que apenas leem dados do servidor. Geralmente usamos requisições *POST* quando o usuário precisa submeter informações por meio de um formulário. Especificaremos o método POST para processar todos os nossos formulários. (Existem mais alguns tipos de requisições, mas não os usaremos neste projeto.)

A função `new_topic()` aceita o objeto de requisição como parâmetro. Quando o usuário inicialmente solicita essa página, o navegador envia uma requisição GET. Depois que o usuário tiver preenchido e submetido o formulário, o navegador enviará uma requisição POST. Conforme a requisição, saberemos se o usuário está solicitando um formulário em

branco (uma requisição GET) ou nos pedindo para processar um formulário preenchido (uma requisição POST).

O teste em u determina se o método de requisição é GET ou POST. Se o método de requisição não for um POST, a requisição provavelmente é um GET, portanto precisamos devolver um formulário em branco (se for outro tipo de requisição, continua sendo seguro devolver um formulário em branco). Criamos uma instância de `TopicForm` v, armazenamos essa instância em uma variável `form` e enviamos o formulário para o template no dicionário de contexto {. Como não incluímos nenhum argumento ao instanciar `TopicForm`, o Django cria um formulário em branco que o usuário poderá preencher.

Se o método de requisição for um POST, o bloco `else` executará e processará os dados submetidos no formulário. Criamos uma instância de `TopicForm` w e passamos os dados fornecidos pelo usuário, armazenados em `request.POST`. O objeto `form` devolvido contém as informações submetidas pelo usuário.

Não podemos salvar as informações submetidas no banco de dados antes de verificar se são válidas x. A função `is_valid()` verifica se todos os campos necessários foram preenchidos (todos os campos em um formulário são obrigatórios por padrão) e se os dados fornecidos são do tipo esperado para o campo – por exemplo, se o tamanho de `text` é menor que 200 caracteres, conforme especificado em *models.py* no Capítulo 18. Essa validação automática nos poupa de muito trabalho. Se tudo estiver válido, chamamos `save()` y, que grava os dados do formulário no banco de dados. Depois que os dados forem salvos, podemos sair dessa página. Usamos `reverse()` para obter o URL da página `topics` e o passamos para `HttpResponseRedirect()` z, que redireciona o navegador do usuário para essa página. Na página `topics`, o usuário deverá ver o assunto que ele acabou de inserir na lista de assuntos.

Template para new_topic

Agora vamos criar um novo template chamado *new_topic.html* para exibir o formulário que acabamos de criar:

new_topic.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}
    <p>Add a new topic:</p>

u   <form action="{% url 'learning_logs:new_topic' %}" method='post'>
v     {% csrf_token %}
w     {{ form.as_p }}
x     <button name="submit">add topic</button>
    </form>

  {% endblock content %}
```

Esse template estende *base.html*, portanto tem a mesma estrutura básica que o restante das páginas de Learning Log. Em u definimos um formulário HTML. O argumento `action` diz ao servidor para qual lugar devem ser enviados os dados submetidos no formulário; nesse caso, eles serão enviados para a função de view `new_topic()`. O argumento `method` diz ao navegador para submeter os dados com uma requisição POST.

O Django utiliza a tag de template `{% csrf_token %}` v para evitar que invasores usem o formulário a fim de ter acesso não autorizado ao servidor (esse tipo de ataque é conhecido como *cross-site request forgery*). Em w exibimos o formulário; nesse ponto, você pode ver como o Django simplifica a execução de tarefas como a exibição de um formulário. Basta incluir a variável de template `{{ form.as_p }}` para que o framework crie todos os campos necessários e exiba o formulário automaticamente. O modificador `as_p` diz a Django para renderizar todos os elementos do formulário em formato de parágrafo, que é uma maneira simples de exibir o formulário de modo organizado.

O Django não cria um botão de submissão para os formulários, portanto definimos um em x.

Criando um link para a página new_topic

A seguir, incluímos um link para a página `new_topic` na página `topics`:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

  <ul>
    --trecho omitido--
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Add a new topic:</a>

{% endblock content %}
```

Coloque o link depois da lista de assuntos existentes. A Figura 19.1 mostra o formulário resultante. Vá em frente e use o formulário para adicionar alguns assuntos novos.

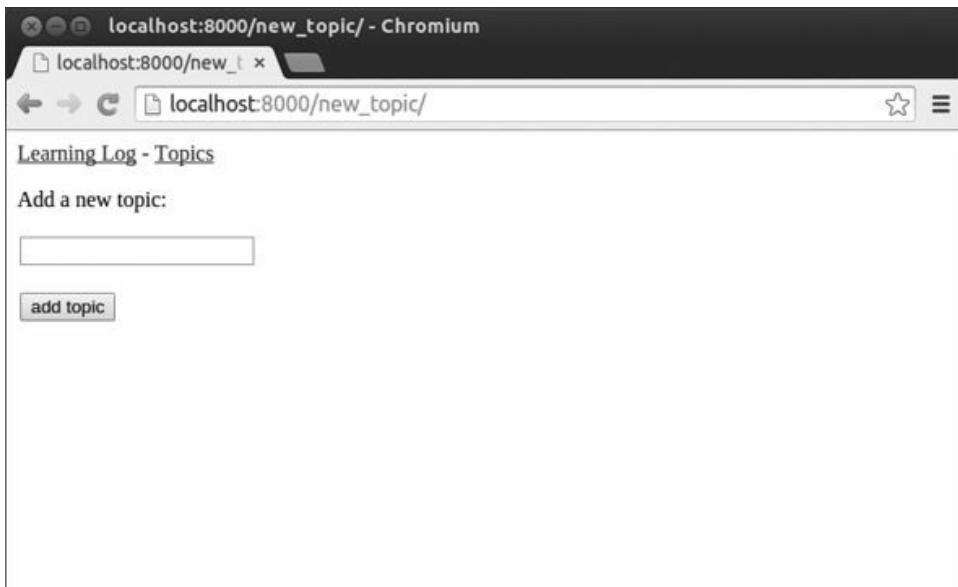



Figura 19.1 – A página para adicionar um novo assunto.

Adicionando novas entradas

Agora que o usuário é capaz de adicionar um novo assunto, ele vai querer acrescentar novas entradas também. Novamente definiremos um URL,

escreveremos uma função de view e um template e criaremos um link para a página. Antes disso, porém, acrescentaremos outra classe em *forms.py*.

ModelForm para entradas

Devemos criar um formulário associado ao modelo `Entry`, porém, desta vez, ele será um pouco mais personalizado que `TopicForm`:

forms.py

```
  from django import forms

  from .models import Topic, Entry

  class TopicForm(forms.ModelForm):
      --trecho omitido--

  class EntryForm(forms.ModelForm):
      class Meta:
          model = Entry
          fields = ['text']
u         labels = {'text': ''}
v         widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

Inicialmente atualizamos a instrução `import` para incluir `Entry`, além de `Topic`. A nova classe `EntryForm` herda de `forms.ModelForm` e tem uma classe `Meta` aninhada que lista o modelo no qual ela está baseada e o campo a ser incluído no formulário. Novamente especificamos um rótulo vazio para o campo `'text'` u.

Em v incluímos o atributo `widgets`. Um *widget* é um elemento de formulário HTML, por exemplo, uma caixa de texto de uma única linha, uma área de texto com várias linhas ou uma lista suspensa. Ao incluir o atributo `widgets`, podemos sobrescrever as opções default de widgets de Django. Ao dizer a Django para usar um elemento `forms.Textarea`, estamos personalizando o widget de entrada para o campo `'text'` de modo que a área de texto tenha 80 colunas, em vez de usar o default de 40. Isso dará espaço suficiente aos usuários para redigir uma entrada significativa.

URL new_entry

Devemos incluir um argumento `topic_id` no URL para adicionar uma nova entrada, pois ela deve estar associada a um assunto em particular. A seguir vemos o URL que acrescentamos em *learning_logs/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    --trecho omitido--
    # Página para adicionar uma nova entrada
    url(r'^new_entry/(?P<topic_id>\d+)/$', views.new_entry,
name='new_entry'),
]
```

Esse padrão de URL corresponde a qualquer URL no formato `http://localhost:8000/new_entry/`*`id`*`/`, em que *`id`* é o número de ID de um assunto. O código `(?P<topic_id>\d+)` captura um valor numérico e o armazena na variável `topic_id`. Quando um URL correspondente a esse padrão for solicitado, o Django enviará a requisição e o ID do assunto para a função de view `new_entry()`.

Função de view new_entry()

A função de view para `new_entry` é bem parecida com a função para adicionar um novo assunto:

views.py

```
from django.shortcuts import render
--trecho omitido--

from .models import Topic
from .forms import TopicForm, EntryForm

--trecho omitido--
def new_entry(request, topic_id):
    """Acrescenta uma nova entrada para um assunto em particular."""
u     topic = Topic.objects.get(id=topic_id)

v     if request.method != 'POST':
          # Nenhum dado submetido; cria um formulário em branco
```

```
w         form = EntryForm()
      else:
          # Dados de POST submetidos; processa os dados
x         form = EntryForm(data=request.POST)
          if form.is_valid():
y             new_entry = form.save(commit=False)
z             new_entry.topic = topic
              new_entry.save()
{             return HttpResponseRedirect(reverse('learning_logs:topic',
                                          args=[topic_id]))

      context = {'topic': topic, 'form': form}
      return render(request, 'learning_logs/new_entry.html', context)
```

Atualizamos a instrução `import` para incluir o `EntryForm` que acabamos de criar. A definição de `new_entry()` tem um parâmetro `topic_id` para armazenar o valor recebido do URL. Precisaremos do assunto para renderizar a página e processar os dados do formulário, portanto utilizamos `topic_id` para obter o objeto correto para o assunto em u.

Em v verificamos se o método de requisição é POST ou GET. O bloco `if` executará se for uma requisição GET, e criaremos uma instância em branco de `EntryForm` w. Se o método de requisição for um POST, processaremos os dados criando uma instância de `EntryForm`, preenchida com os dados de POST do objeto `request` x. Então verificamos se o formulário é válido. Se for, devemos definir o atributo `topic` do objeto de entrada antes de salvá-lo no banco de dados.

Quando chamamos `save()`, incluímos o argumento `commit=False` y para dizer a Django que crie um novo objeto de entrada e o armazene em `new_entry` sem salvá-lo no banco de dados por enquanto. Definimos o atributo `topic` de `new_entry` com o assunto extraído do banco de dados no início da função z; então chamamos `save()` sem argumentos. Essa instrução salva a entrada no banco de dados com o assunto correto associado.

Em { redirecionamos o usuário para a página do assunto. A chamada a `reverse()` exige dois argumentos: o nome do padrão de URL para o qual queremos gerar um URL e uma lista `args` contendo qualquer argumento que deva ser incluído no URL. A lista `args` contém um item: `topic_id`. A

chamada a `HttpResponseRedirect()` então redireciona o usuário para a página do assunto para o qual uma entrada foi criada, e a nova entrada deverá ser vista na lista de entradas.

Template para new_entry

Como podemos ver no código a seguir, o template para `new_entry` é semelhante ao template para `new_topic`:

new_entry.html

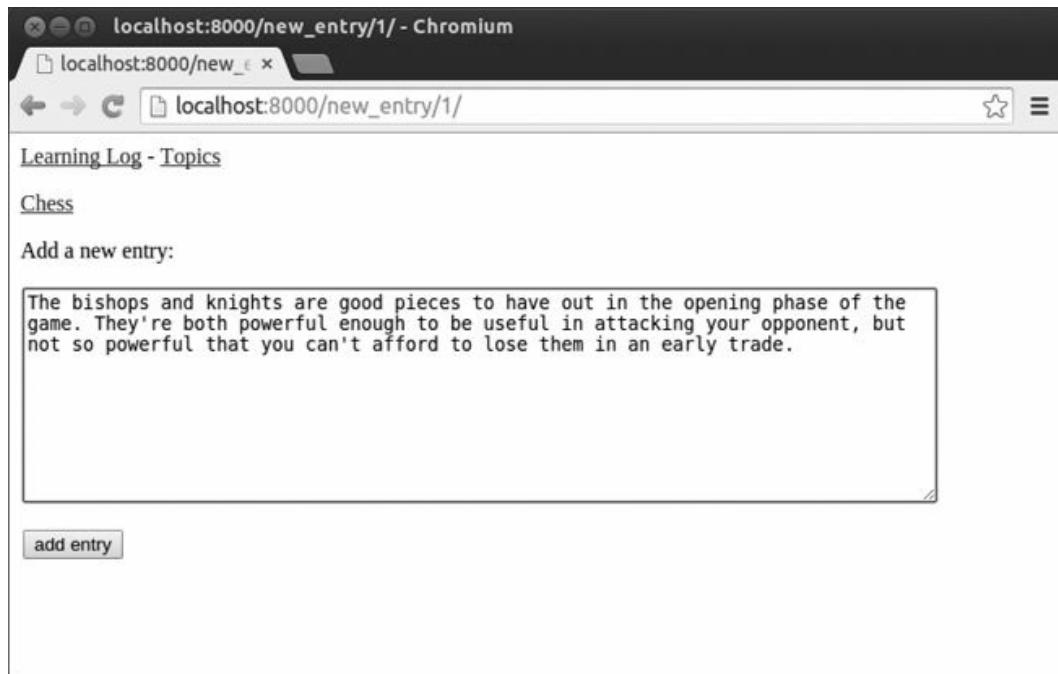
```
  {% extends "learning_logs/base.html" %}

  {% block content %}

u   <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
</p>

    <p>Add a new entry:</p>
v   <form action="{% url 'learning_logs:new_entry' topic.id %}"
method='post'>
      {% csrf_token %}
      {{ form.as_p }}
      <button name='submit'>add entry</button>
    </form>

  {% endblock content %}
```

Mostramos o assunto na parte superior da página **u** para que o usuário possa ver para qual assunto ele está adicionando uma entrada. Esse item também atua como um link de volta para a página principal desse assunto.

O argumento `action` do formulário inclui o valor `topic_id` no URL para que a função de view possa associar a nova entrada ao assunto correto **v**. Exceto por isso, esse template se parece exatamente com *new_topic.html*.

Criando um link para a página new_entry

A seguir precisamos incluir um link para a página `new_entry` a partir da página de cada assunto:

topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">add new
entry</a>
  </p>
  <ul>
  --trecho omitido--
  </ul>

{% endblock content %}
```

Adicionamos o link imediatamente antes de mostrar as entradas, pois acrescentar uma nova entrada será a ação mais comum nessa página. A Figura 19.2 mostra a página `new_entry`. Agora os usuários poderão acrescentar novos assuntos e quantas entradas quiserem para cada assunto. Teste a página `new_entry` adicionando algumas entradas para os assuntos que você criou.

Figura 19.2 – A página `new_entry`.

Editando as entradas

Vamos agora criar uma página que permita aos usuários editar as entradas que já foram adicionadas.

URL edit_entry

O URL para a página deve passar o ID da entrada a ser editada. Eis o código em *learning_logs/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    --trecho omitido--
    # Página para editar uma entrada
    url(r'^edit_entry/(?P<entry_id>\d+)/$', views.edit_entry,
        name='edit_entry'),
]
```

O ID passado no URL (por exemplo, *http://localhost:8000/edit_entry/1/*) é armazenado no parâmetro `entry_id`. O padrão de URL envia requisições que correspondam a esse formato para a função de view `edit_entry()`.

Função de view edit_entry()

Quando a página `edit_entry` receber uma requisição GET, `edit_entry()` devolverá um formulário para editar a entrada. Quando a página receber uma requisição POST com um texto de entrada revisado, a função salvará o texto modificado no banco de dados:

views.py

```
from django.shortcuts import render
--trecho omitido--

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
--trecho omitido--
```

```
  def edit_entry(request, entry_id):
      """Edita uma entrada existente."""
u     entry = Entry.objects.get(id=entry_id)
      topic = entry.topic

      if request.method != 'POST':
          # Requisição inicial; preenche previamente o formulário com a
  entrada atual
v         form = EntryForm(instance=entry)
      else:
          # Dados de POST submetidos; processa os dados
w         form = EntryForm(instance=entry, data=request.POST)
          if form.is_valid():
x             form.save()
y             return HttpResponseRedirect(reverse('learning_logs:topic',
                                          args=[topic.id]))

      context = {'entry': entry, 'topic': topic, 'form': form}
      return render(request, 'learning_logs/edit_entry.html', context)
```

Inicialmente devemos importar o modelo `Entry`. Em u adquirimos o objeto da entrada que o usuário quer editar e o assunto associado a essa entrada. No bloco `if`, executado para uma requisição GET, criamos uma instância de `EntryForm` com o argumento `instance=entry` v. Esse argumento diz a Django para criar o formulário previamente preenchido com informações do objeto de entrada existente. O usuário verá os dados existentes e poderá editá-los.

Ao processar uma requisição POST, passamos os argumentos `instance=entry` e `data=request.POST` w para dizer a Django que crie uma instância de formulário baseada nas informações associadas ao objeto de entrada existente, atualizadas com qualquer dado relevante de `request.POST`. Então verificamos se o formulário é válido; em caso afirmativo, chamamos `save()` sem argumentos x. Em seguida redirecionamos o usuário para a página `topic` y, na qual ele deverá ver a versão atualizada da entrada editada.

Template para edit_entry

Eis o conteúdo de *edit_entry.html*, que é semelhante ao de *new_entry.html*:

edit_entry.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

    <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
  </p>

    <p>Edit entry:</p>

u   <form action="{% url 'learning_logs:edit_entry' entry.id %}"
method='post'>
      {% csrf_token %}
      {{ form.as_p }}
v     <button name="submit">save changes</button>
    </form>

  {% endblock content %}
```

Em u o argumento `action` envia o formulário de volta à função `edit_entry()` para processamento. Incluímos o ID da entrada como argumento na tag `{% url %}` para que a função de view possa modificar o objeto de entrada correto. Colocamos o rótulo *save changes* (salvar alterações) no botão de submissão para lembrar o usuário que ele está salvando alterações, e não criando uma nova entrada v.

Criando um link para a página edit_entry

Agora precisamos incluir um link para a página `edit_entry` para cada entrada na página de um assunto:

topic.html

```
  --trecho omitido--
    {% for entry in entries %}
      <li>
        <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
        <p>{{ entry.text|linebreaks }}</p>
        <p>
```

```
        <a href="{% url 'learning_logs:edit_entry' entry.id %}">edit
entry</a>
      </p>
    </li>
--trecho omitido--
```

Incluímos o link de edição depois que a data de cada entrada e o texto forem exibidos. Usamos a tag de template `{% url %}` para determinar o URL referente ao padrão de URL, cujo nome é `edit_entry`, juntamente com o atributo ID da entrada atual no laço (`entry.id`). O texto do link `"edit entry"` aparece depois de cada entrada na página. A Figura 19.3 mostra a aparência da página de um assunto com esses links.

O projeto Learning Log agora tem a maior parte das funcionalidades necessárias. Os usuários podem acrescentar assuntos e entradas e ler qualquer conjunto de entradas que quiserem. Na próxima seção implementaremos um sistema de registro de usuários para que qualquer pessoa possa ter uma conta em Learning Log e criar o seu próprio conjunto de assuntos e entradas.

Figura 19.3 – Cada entrada agora tem um link para editá-la.

FAÇA VOCÊ MESMO

19.1 – Blog: Inicie um novo projeto Django chamado *Blog*. Crie uma aplicação chamada *blogs* no projeto, com um modelo de nome `BlogPost`. O modelo deve ter campos como `title`, `text` e `date_added`. Crie um superusuário para o projeto e use o site de administração para inserir algumas postagens pequenas. Crie uma página inicial que mostre todos as postagens em ordem cronológica.

Construa um formulário para criar novas postagens e outro para editar postagens existentes. Preencha seus formulários para garantir que funcionem.

Criando contas de usuário

Nesta seção implementaremos um sistema de cadastro e de autorização de usuários para permitir que as pessoas registrem uma conta e façam login e logout. Criaremos uma nova aplicação que contenha todas as funcionalidades relacionadas à manipulação de usuários. Também modificaremos um pouco o modelo `Topic` para que todo assunto pertença a um usuário em particular.

Aplicação users

Começaremos criando uma nova aplicação chamada `users` utilizando o comando `startapp`:

```
  (ll_env)learning_log$ python manage.py startapp users
  (ll_env)learning_log$ ls
u db.sqlite3  learning_log  learning_logs  ll_env  manage.py  users
  (ll_env)learning_log$ ls users
v admin.py __init__.py migrations models.py tests.py views.py
```

Esse comando cria um novo diretório chamado *users* **u**, com uma estrutura idêntica àquela da aplicação `learning_logs` **v**.

Adicionando users em settings.py

Precisamos adicionar nossa nova aplicação em `INSTALLED_APPS`, no arquivo *settings.py*, desta maneira:

settings.py

```
--trecho omitido--
INSTALLED_APPS = (
    --trecho omitido--
```

```
    # Minhas aplicações
    'learning_logs',
    'users',
)
--trecho omitido--
```

Agora Django incluirá a aplicação `users` no projeto como um todo.

Incluindo os URLs de users

Em seguida precisamos modificar o *urls.py* raiz para que inclua os URLs que utilizaremos na aplicação `users`:

urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls', namespace='users')),
    url(r'', include('learning_logs.urls', namespace='learning_logs')),
]
```

Acrescentamos uma linha para incluir o arquivo *urls.py* de `users`. Essa linha corresponderá a qualquer URL que comece com a palavra *users*, por exemplo, *http://localhost:8000/users/login/*. Também criamos o namespace `'users'` para podermos distinguir os URLs pertencentes à aplicação `learning_logs` daqueles que pertencem à aplicação `users`.

Página de login

Em primeiro lugar, vamos implementar uma página de login. Usaremos a view `login` default disponibilizada por Django, portanto o padrão de URL será um pouco diferente. Crie um novo arquivo *urls.py* no diretório *learning_log/users/* e acrescente o seguinte nesse arquivo:

urls.py

```
"""Define padrões de URL para users"""

from django.conf.urls import url
```

```
u from django.contrib.auth.views import login

  from . import views

  urlpatterns = [
      # Página de login
v     url(r'^login/$', login, {'template_name': 'users/login.html'},
          name='login'),
  ]
```

Inicialmente importamos a view `login` default u. O padrão da página de login corresponde ao URL *http://localhost:8000/users/login/* v. Quando Django lê esse URL, a palavra *users* diz a ele para consultar *users/urls.py*, e *login* lhe diz para enviar requisições à view `login` default de Django (observe que o argumento da view é `login`, e não `views.login`). Como não estamos escrevendo nossa própria função de view, passamos um dicionário que diz a Django em que lugar ele poderá encontrar o template que estamos prestes a escrever. Esse template fará parte da aplicação `users`, e não de `learning_logs`.

Template de login

Quando o usuário solicitar a página de login, o Django usará sua view `login` default, mas ainda precisamos fornecer um template para a página. No diretório *learning_log/users/*, crie um diretório chamado *templates*; dentro dele crie outro diretório de nome *users*. Eis o template *login.html*, que você deverá salvar em *learning_log/users/templates/users/*:

login.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

u   {% if form.errors %}
    <p>Your username and password didn't match. Please try again.</p>
    {% endif %}

v   <form method="post" action="{% url 'users:login' %}">
    {% csrf_token %}
```

```
w   {{ form.as_p }}

x   <button name="submit">log in</button>
y   <input type="hidden" name="next" value="{% url 'learning_logs:index'
%}" />
    </form>

  {% endblock content %}
```

Esse template estende *base.html* para garantir que a página de login tenha a mesma aparência do restante do site. Observe que um template em uma aplicação é capaz de estender um template de outra aplicação.

Se o atributo `errors` do formulário estiver definido, exibiremos uma mensagem de erro u informando que a combinação do nome do usuário e a senha não corresponde a nenhum dado armazenado no banco de dados.

Queremos que a view de login processe o formulário, portanto definimos o argumento `action` com o URL da página de login v. A view de login envia um formulário para o template, e cabe a nós exibir o formulário w e acrescentar um botão de submissão x. Em y incluímos um elemento de formulário oculto, `'next'`; o argumento `value` diz a Django para onde o usuário deve ser redirecionado depois que tiver feito login com sucesso. Nesse caso, enviamos o usuário de volta à página inicial.

Criando um link para a página de login

Vamos adicionar o link para login em *base.html* para que ele apareça em todas as páginas. Não queremos que o link seja exibido quando o usuário já estiver logado, portanto vamos inseri-lo em uma tag `{% if %}`:

base.html

```
  <p>
    <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
    <a href="{% url 'learning_logs:topics' %}">Topics</a> -
u   {% if user.is_authenticated %}
v     Hello, {{ user.username }}.
    {% else %}
w      <a href="{% url 'users:login' %}">log in</a>
    {% endif %}
```

```
</p>

{% block content %}{% endblock content %}
```

No sistema de autenticação de Django, todo template tem uma variável `user` disponível, que sempre tem um atributo `is_authenticated` definido: o atributo será `True` se o usuário estiver logado, e `False` se não o estiver. Isso permite que você mostre uma mensagem aos usuários autenticados e outra para os usuários não autenticados.

Nesse exemplo exibimos uma saudação aos usuários logados no momento **u**. Usuários autenticados têm um atributo `username` adicional definido, que usamos para personalizar a saudação e lembrar o usuário de que ele está logado **v**. Em **w** exibimos um link para a página de login para os usuários que não tenham sido autenticados.

Usando a página de login

Já criamos uma conta de usuário, portanto vamos fazer login para ver se a página funciona. Acesse *http://localhost:8000/admin/*. Se você continua logado como administrador, procure um link de logout no cabeçalho e clique nesse link.

Depois de fazer logout, acesse *http://localhost:8000/users/login/*. Você deverá ver uma página de login semelhante àquela mostrada na Figura 19.4. Forneça o nome de usuário e a senha criados anteriormente, e você deverá retornar à página de índice. O cabeçalho na página inicial deve exibir uma saudação personalizada com o seu nome de usuário.

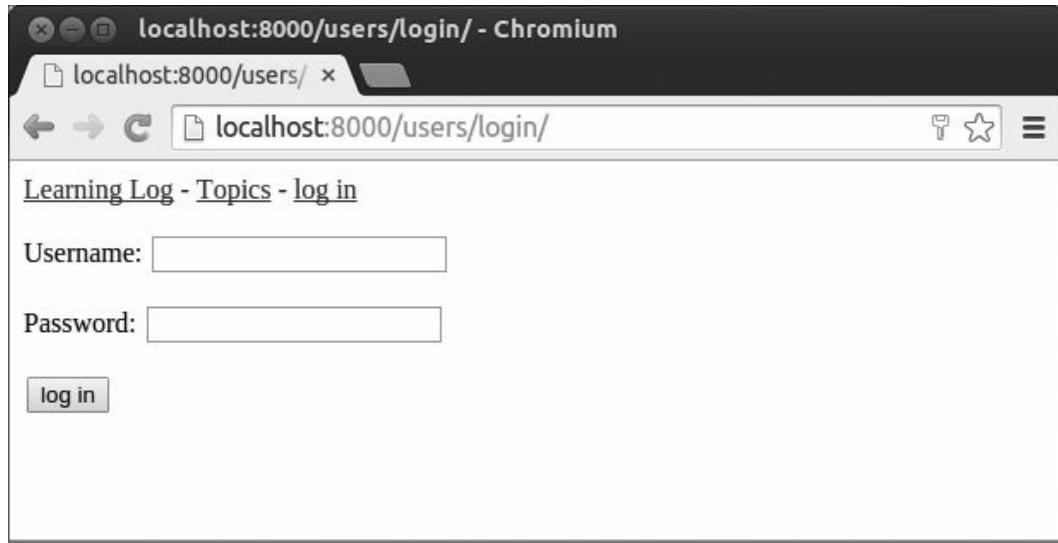



Figura 19.4 – A página de login.

Fazendo logout

Agora devemos oferecer uma maneira de os usuários fazerem logout. Não criaremos uma página para logout; os usuários simplesmente clicarão em um link e serão enviados de volta à página inicial. Definiremos um padrão de URL para o link de logout, escreveremos uma função de view e forneceremos um link para logout em *base.html*.

URL para logout

O código a seguir define o padrão de URL para logout, que corresponde ao URL *http://localhost:8000/users/logout/*. Eis o código em *users/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    # Página de login
    --trecho omitido--
    # Página de logout
    url(r'^logout/$', views.logout_view, name='logout'),
]
```

O padrão de URL envia a requisição para a função `logout_view()`; ela recebe esse nome para distingui-la da função `logout()`, que chamaremos a partir da view. (Certifique-se de que você esteja modificando *users/urls.py*,

e não *learning_log/urls.py*.)

Função de view logout_view()

A função `logout_view()` é simples: simplesmente importamos a função `logout()` de Django, chamamos essa função e então redirecionamos o usuário para a página inicial. Abra *users/views.py* e insira o código a seguir:

views.py

```
  from django.http import HttpResponseRedirect
  from django.core.urlresolvers import reverse
u from django.contrib.auth import logout

  def logout_view(request):
      """Faz logout do usuário."""
v     logout(request)
w     return HttpResponseRedirect(reverse('learning_logs:index'))
```

Importamos a função `logout()` de `django.contrib.auth` u. Na função `logout_view()`, chamamos `logout()` v, que exige o objeto `request` como argumento. Então redirecionamos o usuário para a página inicial w.

Criando um link para a view de logout

Agora precisamos de um link para logout. Ele será incluído como parte de *base.html* para que esteja disponível em todas as páginas, e estará na parte `{% if user.is_authenticated %}` para que apenas os usuários já logados possam vê-lo:

base.html

```
  --trecho omitido--
    {% if user.is_authenticated %}
      Hello, {{ user.username }}.
      <a href="{% url 'users:logout' %}">log out</a>
    {% else %}
      <a href="{% url 'users:login' %}">log in</a>
    {% endif %}
  --trecho omitido--
```

A Figura 19.5 mostra a página inicial atual conforme ela aparece para um

usuário logado. A estilização é mínima porque nosso foco está em criar um site que funcione de forma apropriada. Quando o conjunto necessário de funcionalidades estiver pronto, estilizaremos o site para que tenha uma aparência mais profissional.

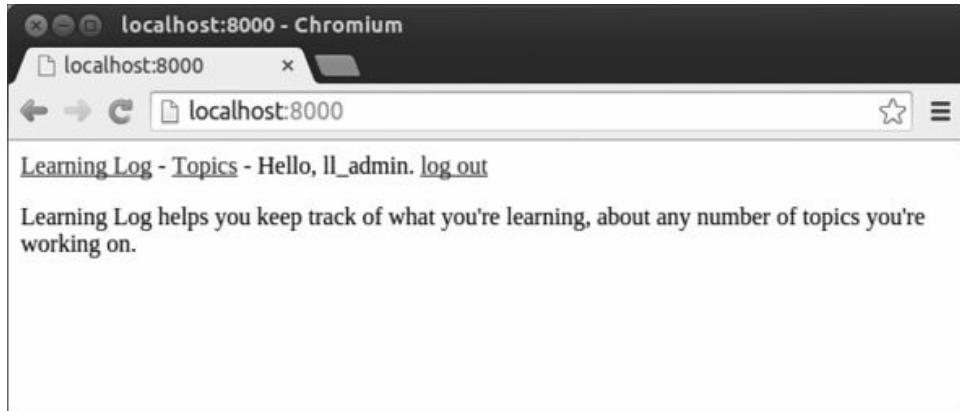



Figura 19.5 – A página inicial com uma saudação personalizada e um link para logout.

Página de cadastro

Em seguida vamos criar uma página que permita que novos usuários se registrem. Usaremos o `UserCreationForm` default de Django, mas escreveremos nossa própria função de view e o template.

URL register

O código a seguir fornece o padrão de URL para a página de cadastro de usuários, novamente em *users/urls.py*:

urls.py

```
--trecho omitido--
urlpatterns = [
    # Página de login
    --trecho omitido--
    # Página de cadastro
    url(r'^register/$', views.register, name='register'),
]
```

Esse padrão corresponde ao URL *http://localhost:8000/users/register/* e envia requisições para a função `register()` que estamos prestes a escrever.

Função de view register()

A função de view `register()` deve exibir um formulário de cadastro em branco quando a página de inscrição for solicitada pela primeira vez e então deve processar o formulário de cadastro completo quando esse for submetido. Se um cadastro for bem-sucedido, a função também deverá fazer o login do novo usuário. Acrescente o código a seguir em *users/views.py*:

views.py

```
  from django.shortcuts import render
  from django.http import HttpResponseRedirect
  from django.core.urlresolvers import reverse
  from django.contrib.auth import login, logout, authenticate
  from django.contrib.auth.forms import UserCreationForm

  def logout_view(request):
  --trecho omitido--

  def register(request):
      """Faz o cadastro de um novo usuário."""
      if request.method != 'POST':
          # Exibe o formulário de cadastro em branco
u         form = UserCreationForm()
      else:
          # Processa o formulário preenchido
v         form = UserCreationForm(data=request.POST)

w         if form.is_valid():
x             new_user = form.save()
              # Faz login do usuário e o redireciona para a página
  inicial
y             authenticated_user =
authenticate(username=new_user.username,
                  password=request.POST['password1'])
z             login(request, authenticated_user)
{             return HttpResponseRedirect(reverse('learning_logs:index'))

      context = {'form': form}
      return render(request, 'users/register.html', context)
```

Em primeiro lugar, importamos a função `render()`. Então importamos as funções `login()` e `authenticate()` para fazer login do usuário, se suas informações de cadastro estiverem corretas. Também importamos o `UserCreationForm` default. Na função `register()`, verificamos se estamos respondendo a uma requisição POST. Se não estivermos, criamos uma instância de `UserCreationForm` sem dados iniciais u.

Se estivermos respondendo a uma requisição POST, criamos uma instância de `UserCreationForm` com base nos dados submetidos v. Verificamos se os dados são válidos w – nesse caso, se o nome do usuário tem os caracteres apropriados, se as senhas são iguais e se o usuário não está tentando fazer algo malicioso em sua submissão.

Se os dados submetidos forem válidos, chamamos o método `save()` do formulário para salvar o nome do usuário e o hash da senha no banco de dados x. O método `save()` devolve um objeto para o usuário recém-criado, que armazenamos em `new_user`.

Quando as informações do usuário forem salvas, fazemos o seu login, que é um processo de dois passos: chamamos `authenticate()` com os argumentos `new_user.username` e a senha y. Ao se cadastrar, o usuário é solicitado a fornecer duas senhas iguais, e como o formulário é válido sabemos que as senhas são idênticas, portanto podemos usar qualquer uma delas. Nesse caso, lemos o valor associado à chave `'password1'` nos dados de POST do formulário. Se o nome do usuário e a senha estiverem corretos, o método devolverá um objeto com o usuário autenticado, que armazenaremos em `authenticated_user`. Então chamamos a função `login()` com os objetos `request` e `authenticated_user` z, o que criará uma sessão válida para o nome do usuário. Por fim, redirecionamos o usuário para a página inicial {, na qual uma saudação personalizada no cabeçalho informa que o cadastro foi bem-sucedido.

Template de cadastro

O template para a página de cadastro de usuários é semelhante ao da página de login. Lembre-se de salvá-lo no mesmo diretório em que está *login.html*:

register.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">register</button>
    <input type="hidden" name="next" value="{% url
'learning_logs:index' %}" />
  </form>

{% endblock content %}
```

Usamos o método `as_p` novamente para que Django exiba todos os campos do formulário de modo apropriado, incluindo qualquer mensagem de erro caso o formulário não seja preenchido corretamente.

Criando um link para a página de cadastro

A seguir acrescentaremos o código para mostrar o link para a página de cadastro a qualquer usuário que não esteja logado no momento:

base.html

```
--trecho omitido--
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">log out</a>
  {% else %}
    <a href="{% url 'users:register' %}">register</a> -
    <a href="{% url 'users:login' %}">log in</a>
  {% endif %}
--trecho omitido--
```

Agora os usuários logados verão uma saudação personalizada e um link para logout. Usuários não logados verão um link para a página de cadastro e um link para login. Teste a página de cadastro criando várias contas de

usuários com nomes diferentes.

Na próxima seção vamos restringir algumas das páginas para que elas estejam disponíveis somente aos usuários cadastrados e garantir que todos os assuntos pertençam a um usuário específico.

NOTA O sistema de cadastro que criamos permite que qualquer pessoa crie inúmeras contas em Learning Log. Porém, alguns sistemas exigem que os usuários confirmem sua identidade enviando-lhes um email de confirmação que o usuário deve responder. Ao fazer isso, o sistema gera menos contas spam que o sistema simples que estamos utilizando aqui. No entanto, enquanto estiver aprendendo a criar aplicações, é perfeitamente apropriado treinar com um sistema simples de cadastro de usuários como o que estamos usando.

FAÇA VOCÊ MESMO

19.2 – Contas no blog: Adicione um sistema de autenticação e cadastro de usuários no projeto Blog iniciado no Exercício 19.1 (página 556). Garanta que usuários logados vejam seus nomes de usuário em algum lugar da tela, enquanto usuários não cadastrados vejam um link para a página de cadastro.

Permitindo que os usuários tenham seus próprios dados

Os usuários devem ser capazes de inserir dados exclusivos a eles, portanto criaremos um sistema para descobrir quais dados pertencem a quais usuários e então restringiremos o acesso a determinadas páginas para que os usuários possam trabalhar apenas com seus próprios dados.

Nesta seção modificaremos o modelo `Topic` para que cada assunto pertença a um usuário específico. Isso envolverá as entradas também, pois toda entrada pertence a um assunto específico. Começaremos restringindo o acesso a determinadas páginas.

Restringindo o acesso com @login_required

Django facilita restringir o acesso a determinadas páginas aos usuários logados por meio do decorador `@login_required`. Um *decorador* é uma diretiva colocada imediatamente antes da definição de uma função, que Python aplica a ela antes que seja executada a fim de alterar o modo como essa função se comporta. Vamos ver um exemplo.

Restringindo o acesso à página de assuntos

Todo assunto pertencerá a um usuário, portanto apenas usuários cadastrados deverão ser capazes de solicitar a página de assuntos. Acrescente o código a seguir em *learning_logs/views.py*:

views.py

```
--trecho omitido--
from django.core.urlresolvers import reverse
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--trecho omitido--

@login_required
def topics(request):
    """Mostra todos os assuntos."""
    --trecho omitido--
```

Inicialmente importamos a função `login_required()`. Aplicamos `login_required()` como um decorador da função de view `topics()` prefixando `login_required` com o símbolo `@` para que Python saiba que deve executar o código em `login_required()` antes do código em `topics()`.

O código em `login_required()` verifica se um usuário está logado, e Django executará o código em `topics()` somente em caso afirmativo. Se não estiver logado, o usuário será redirecionado para a página de login.

Para fazer esse redirecionamento funcionar, devemos modificar *settings.py* para que Django saiba em que local pode encontrar a página de login. Acrescente o código a seguir no final de *settings.py*:

settings.py

```
"""
Configurações de Django para o projeto learning_log
--trecho omitido--

# Minhas configurações
LOGIN_URL = '/users/login/'
```

Agora, quando um usuário não autenticado solicitar uma página protegida pelo decorador `@login_required`, Django enviará o usuário para o URL definido por `LOGIN_URL` em *settings.py*.

Você pode testar essa configuração fazendo logout de qualquer conta de usuário e acessando a página inicial. Em seguida clique no link Topics, que deverá redirecionar você para a página de login. Então faça login em qualquer uma de suas contas einicial, clique no link Topics novamente. Você deverá ser capaz de acessar a página de assuntos.

Restringindo o acesso em Learning Log

Django facilita restringir o acesso às páginas, mas você deve decidir quais páginas serão protegidas. É melhor pensar em quais páginas não devem ser restritas antes, e então restringir todas as demais páginas do projeto. Podemos corrigir facilmente um excesso de restrição, e isso é menos perigoso que deixar páginas sensíveis sem restrição.

Em Learning Log, manteremos a página inicial, a página de cadastro e o logout sem restrição. Restringiremos o acesso a todas as outras páginas.

Eis o código *de learning_logs/views.py* com os decoradores `@login_required` aplicados a todas as views, exceto em `index()`:

views.py

```
--trecho omitido--
@login_required
def topics(request):
    --trecho omitido--

@login_required
def topic(request, topic_id):
    --trecho omitido--

@login_required
def new_topic(request):
    --trecho omitido--

@login_required
def new_entry(request, topic_id):
```

```
    --trecho omitido--

@login_required
def edit_entry(request, entry_id):
    --trecho omitido--
```

Tente acessar cada uma dessas páginas enquanto não estiver logado; você será redirecionado para a página de login. Você também não será capaz de clicar em links para páginas como `new_topic`. No entanto, se fornecer o URL *http://localhost:8000/new_topic/*, você será redirecionado para a página de login. Restrinja o acesso a qualquer URL publicamente acessível e relacionado a dados privados do usuário.

Associando dados a determinados usuários

Agora precisamos associar os dados submetidos ao usuário que os submeteu. Devemos associar somente os dados do nível mais alto da hierarquia a um usuário; os dados de nível mais baixo acompanharão a associação. Por exemplo, em Learning Log, os assuntos estão no nível mais alto dos dados da aplicação, e todas as entradas estão associadas a um assunto. Desde que cada assunto pertença a um usuário específico, seremos capazes de identificar o dono de cada entrada no banco de dados.

Modificaremos o modelo `Topic` acrescentando um relacionamento de chave estrangeira com um usuário. Então teremos que migrar o banco de dados. Por fim, modificaremos algumas das views para que mostrem apenas os dados associados ao usuário logado no momento.

Modificando o modelo Topic

A modificação em *models.py* envolve apenas duas linhas:

models.py

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """Um assunto sobre o qual o usuário está aprendendo."""
    text = models.CharField(max_length=200)
```

```
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User)

    def __str__(self):
        """Devolve uma representação em string do modelo."""
        return self.text

class Entry(models.Model):
    --trecho omitido--
```

Inicialmente importamos o modelo `User` de `django.contrib.auth`. Então adicionamos um campo `owner` em `Topic`, que determina um relacionamento de chave estrangeira com o modelo `User`.

Identificando usuários existentes

Quando migramos o banco de dados, Django o modificará para que ele possa armazenar uma associação entre cada assunto e um usuário. Para fazer a migração, o framework precisa saber qual usuário deve ser associado a cada assunto existente. A abordagem mais simples é associar todos os assuntos existentes a um usuário – por exemplo, o superusuário. Em primeiro lugar, precisamos conhecer o ID desse usuário.

Vamos ver os IDs de todos os usuários criados até agora. Inicie uma sessão de shell de Django e execute os comandos a seguir:

```
  (venv)learning_log$ python manage.py shell
u >>> from django.contrib.auth.models import User
v >>> User.objects.all()
  [<User: ll_admin>, <User: eric>, <User: willie>]
w >>> for user in User.objects.all():
  ...     print(user.username, user.id)
  ...
  ll_admin 1
  eric 2
  willie 3
  >>>
```

Em **u** importamos o modelo `User` para a sessão de shell. Então vimos todos os usuários criados até agora **v**. A saída mostra três usuários: *ll_admin*, *eric* e *willie*.

Em w percorremos a lista de usuários com um laço e exibimos o nome e o ID de cada usuário. Quando Django perguntar qual usuário deve ser associado aos assuntos existentes, usaremos um desses valores de ID.

Migrando o banco de dados

Agora que já conhecemos os IDs, podemos migrar o banco de dados.

```
u (venv)learning_log$ python manage.py makemigrations learning_logs
v You are trying to add a non-nullable field 'owner' to topic without a
default;
we can't do that (the database needs something to populate existing
rows).
w Please select a fix:
  1) Provide a one-off default now (will be set on all existing rows)
  2) Quit, and let me add a default in models.py
x Select an option: 1
y Please enter the default value now, as valid Python
  The datetime and django.utils.timezone modules are available, so you
  can do e.g. timezone.now()
z >>> 1
  Migrations for 'learning_logs':
    0003_topic_owner.py:
      - Add field owner to topic
```

Começamos executando o comando `makemigrations` u. Na saída em v Django informa que estamos tentando adicionar um campo obrigatório (não nulo) em um modelo existente (`topic`) sem valor default especificado. O framework nos dá duas opções em w: podemos fornecer um default nesse momento ou podemos sair e acrescentar um valor default em *models.py*. Em x escolhemos a primeira opção. O Django então nos pede para fornecer o valor default y.

Para associar todos os assuntos existentes ao usuário administrador original, *ll_admin*, forneci o ID de usuário igual a 1 em z. Você pode usar o ID de qualquer usuário criado; não precisa ser um superusuário. O Django então migra o banco de dados usando esse valor e gera o arquivo de migração *0003_topic_owner.py*, que acrescenta o campo `owner` ao modelo `Topic`.

Agora podemos continuar a migração. Execute o seguinte em um

ambiente virtual ativo:

```
  (venv)learning_log$ python manage.py migrate
  Operations to perform:
    Synchronize unmigrated apps: messages, staticfiles
    Apply all migrations: learning_logs, contenttypes, sessions, admin,
  auth
  --trecho omitido--
  Running migrations:
    Rendering model states... DONE
u   Applying learning_logs.0003_topic_owner... OK
  (venv)learning_log$
```

O Django aplica a nova migração e o resultado é `OK` u.

Podemos conferir se a migração funcionou conforme esperado na sessão de shell, assim:

```
u >>> from learning_logs.models import Topic
v >>> for topic in Topic.objects.all():
  ...     print(topic, topic.owner)
  ...
  Chess ll_admin
  Rock Climbing ll_admin
  >>>
```

Importamos `Topic` de `learning_logs.models` u e então percorremos todos os assuntos existentes em um laço, exibindo cada um deles e o usuário ao qual ele pertence v. Você pode ver que todo assunto agora pertence ao usuário *ll_admin*.

NOTA Podemos simplesmente reiniciar o banco de dados em vez de fazer uma migração, mas perderíamos todos os dados existentes. Aprender a migrar um banco de dados ao mesmo tempo em que a integridade dos dados dos usuários é mantida é uma boa prática. Se quiser recomeçar com um banco de dados limpo, execute o comando `python manage.py flush` para recriar o banco de dados. Você deverá criar um novo superusuário, e todos os seus dados serão perdidos.

Restringindo o acesso aos assuntos para os usuários apropriados

No momento, se você estiver logado poderá ver todos os assuntos, independentemente do usuário com que estiver logado. Mudaremos isso

mostrando aos usuários somente os assuntos que lhes pertencem.

Faça a seguinte alteração na função `topics()` em *views.py*:

views.py

```
--trecho omitido--
@login_required
def topics(request):
    """Mostra todos os assuntos."""
    topics =
Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--trecho omitido--
```

Quando um usuário está logado, o objeto de requisição tem um atributo `request.user` definido, que armazena informações sobre o usuário. O fragmento de código `Topic.objects.filter(owner=request.user)` diz a Django para recuperar apenas os objetos `Topic` do banco de dados cujo atributo `owner` seja igual ao usuário atual. Como não estamos mudando o modo como os assuntos são exibidos, não é necessário alterar o template da página de assuntos.

Para ver se isso funciona, faça login com o usuário ao qual você associou todos os assuntos existentes e acesse a página de assuntos. Você deverá ver todos os assuntos. Agora faça logout e login novamente com um usuário diferente. A página de assuntos não deverá listar nenhum assunto.

Protegendo os assuntos de um usuário

Na verdade, ainda não restringimos o acesso às páginas dos assuntos, portanto qualquer usuário cadastrado poderia experimentar vários URLs, por exemplo, *http://localhost:8000/topics/1/*, e obter as páginas de assuntos que por acaso forem encontradas.

Experimente fazer isso. Enquanto estiver logado com o usuário que é dono de todos os assuntos, copie o URL ou anote o ID do URL de um assunto, faça logout e depois faça login com um usuário diferente. Forneça o URL desse assunto. Você será capaz de ler as entradas, apesar de estar

logado com um usuário diferente.

Corrigiremos isso agora fazendo uma verificação antes de recuperar as entradas solicitadas na função de view `topic()`:

views.py

```
  from django.shortcuts import render
u from django.http import HttpResponseRedirect, Http404
  from django.core.urlresolvers import reverse
  --trecho omitido--

  @login_required
  def topic(request, topic_id):
      """Mostra um único assunto e todas as suas entradas."""
      topic = Topic.objects.get(id=topic_id)
      # Garante que o assunto pertence ao usuário atual
v     if topic.owner != request.user:
          raise Http404

      entries = topic.entry_set.order_by('-date_added')
      context = {'topic': topic, 'entries': entries}
      return render(request, 'learning_logs/topic.html', context)
  --trecho omitido--
```

Uma resposta 404 é uma resposta de erro padrão, devolvida quando um recurso requisitado não existe em um servidor. Em nosso caso, importamos a exceção `Http404` u, que levantaremos se o usuário requisitar um assunto que não deva ver. Depois de receber uma requisição de assunto, garantimos que o usuário associado ao assunto corresponda ao usuário logado no momento, antes de renderizar a página. Se o usuário atual não for o dono do assunto requisitado, levantaremos a exceção `Http404` v e Django devolverá uma página de erro 404.

Se tentar visualizar as entradas de um assunto pertencente a outro usuário, você verá uma mensagem *Page Not Found* (Página não encontrada) de Django. No Capítulo 20 configuraremos o projeto para que os usuários vejam uma página de erro apropriada.

Protegendo a página edit_entry

As páginas `edit_entry` têm URLs no formato `http://localhost:8000/edit_entry/`*`id_entrada`*`/`, em que *`id_entrada`* é um número. Vamos proteger essa página para que ninguém possa usar o URL para ter acesso às entradas de outra pessoa:

views.py

```
--trecho omitido--
@login_required
def edit_entry(request, entry_id):
    """Edita uma entrada existente."""
    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        # Requisição inicial; preenche previamente o formulário com a
entrada atual
        --trecho omitido--
```

Recuperamos a entrada e o assunto associado a ela. Então verificamos se o dono do assunto coincide com o usuário logado no momento; se não forem iguais, levantamos uma exceção `Http404`.

Associando novos assuntos ao usuário atual

No momento, nossa página para adição de novos assuntos não está funcionando, pois ela não associa novos assuntos a nenhum usuário em particular. Se você tentar adicionar um novo assunto, verá a mensagem de erro `IntegrityError` juntamente com `learning_logs_topic.user_id may not be NULL` (`learning_logs_topic.user_id` não pode ser NULL). O Django está dizendo que você não pode criar um novo assunto sem especificar um valor para o campo `owner` do assunto.

Há uma correção simples para esse problema, pois temos acesso ao usuário atual por meio do objeto `request`. Adicione o código a seguir, que associa o novo assunto ao usuário atual:

views.py

```
--trecho omitido--
@login_required
def new_topic(request):
    """Adiciona um novo assunto."""
    if request.method != 'POST':
        # Nenhum dado submetido; cria um formulário em branco
        form = TopicForm()
    else:
        # Dados de POST submetidos; processa os dados
        form = TopicForm(request.POST)
        if form.is_valid():
u           new_topic = form.save(commit=False)
v           new_topic.owner = request.user
w           new_topic.save()
            return
HttpResponseRedirect(reverse('learning_logs:topics'))

    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
--trecho omitido--
```

Quando chamamos `form.save()` pela primeira vez, passamos o argumento `commit=False` porque precisamos modificar o novo assunto antes de salvá-lo no banco de dados u. Então definimos o atributo `owner` do novo assunto com o usuário atual v. Por fim, chamamos `save()` na instância do assunto que acabou de ser definido w. Agora o assunto tem todos os dados necessários e será salvo com sucesso.

Você deverá ser capaz de adicionar quantos novos assuntos quiser para quantos usuários diferentes desejar. Cada usuário terá acesso apenas aos seus próprios dados, seja para visualizá-los, para fornecer novos dados ou modificar dados antigos.

FAÇA VOCÊ MESMO

19.3 – Refatoração: Há dois lugares em *views.py* em que garantimos que o usuário associado a um assunto seja igual ao usuário logado no momento. Coloque o código dessa verificação em uma função chamada `check_topic_owner()` e chame essa função nos lugares apropriados.

19.4 – Protegendo new_entry: Um usuário pode adicionar uma nova entrada no

registro de aprendizado de outro usuário se fornecer um URL com o ID de um assunto que pertença a outro usuário. Evite esse ataque verificando se o usuário atual é dono do assunto associado à entrada antes de salvar a nova entrada.

19.5 – Blog protegido: Em seu projeto Blog, garanta que toda postagem de blog esteja associada a um usuário em particular. Certifique-se de que todas as postagens sejam publicamente acessíveis, mas apenas os usuários cadastrados possam adicionar postagens e editar postagens existentes. Na view que permite aos usuários editar suas postagens, garanta que o usuário esteja editando suas próprias postagens antes de processar o formulário.

Resumo

Neste capítulo aprendemos a usar formulários para permitir que os usuários adicionem novos assuntos e entradas, além de editar entradas existentes. Então vimos como implementar contas de usuários. Permitimos que usuários existentes fizessem login e logout e aprendemos a usar o `UserCreationForm` default de Django para que as pessoas pudessem criar novas contas.

Depois de desenvolver um sistema simples de autenticação e de cadastro de usuários, restringimos o acesso a determinadas páginas aos usuários logados usando o decorador `@login_required`. Então atribuímos dados a usuários específicos por meio de um relacionamento de chave estrangeira. Também aprendemos a migrar o banco de dados quando a migração exige que você especifique alguns dados default.

Por fim, vimos como garantir que um usuário possa ver apenas os dados que lhe pertencem, modificando as funções de view. Recuperamos os dados apropriados usando o método `filter()` e aprendemos a comparar o dono do dado solicitado com o usuário logado no momento.

Nem sempre será óbvio de imediato quais dados devem ser disponibilizados e quais devem ser protegidos, mas essa habilidade será adquirida com a prática. As decisões que tomamos neste capítulo para proteger os dados de nossos usuários mostram por que trabalhar com outras pessoas é uma boa ideia quando desenvolvemos um projeto: ter alguém para observar o seu projeto faz com que seja mais provável identificar áreas vulneráveis.

Agora temos um projeto totalmente funcional executando em nosso

computador local. No último capítulo, estilizaremos o projeto Learning Log para deixá-lo visualmente atraente e o implantaremos em um servidor para que qualquer pessoa com acesso à internet possa se cadastrar e criar uma conta.

20

ESTILIZANDO E IMPLANTANDO UMA APLICAÇÃO

O projeto Learning Log está totalmente funcional agora, mas não tem nenhuma estilização e executa somente em seu computador local. Neste capítulo estilizaremos o projeto de forma simples, porém profissional, e então o implantaremos em um servidor ativo para que qualquer pessoa no mundo possa criar uma conta.

Para a estilização, usaremos a biblioteca Bootstrap – uma coleção de ferramentas para estilização de aplicações web para que elas tenham uma aparência profissional em todos os dispositivos modernos, de um grande monitor de tela plana até um smartphone. Para isso, usaremos a aplicação django-bootstrap3, que também fará com que você adquira prática no uso de aplicações desenvolvidas por outros desenvolvedores que utilizem Django.

Faremos a implantação de Learning Log com o Heroku, um site que permite carregar o seu projeto em um de seus servidores, deixando-o disponível a qualquer pessoa com uma conexão de internet. Também começaremos a usar um sistema de controle de versões chamado Git para monitorar as alterações no projeto.

Quando tiver concluído o projeto Learning Log, você será capaz de desenvolver aplicações web simples, deixá-las com uma boa aparência e implantá-las em um servidor ativo. Também poderá usar recursos mais sofisticados de aprendizagem à medida que desenvolver suas habilidades.

Estilizando o Learning Log

Ignoramos a estilização propositalmente até agora para que nos concentrássemos antes nas funcionalidades de Learning Log. Essa é uma boa maneira de abordar o desenvolvimento, pois uma aplicação só será útil se ela funcionar. É claro que depois que estiver funcionando, a aparência da aplicação será essencial para que as pessoas queiram usá-la.

Nesta seção apresentarei a aplicação django-bootstrap3 e mostrarei como integrá-la em um projeto para deixá-lo pronto para implantação.

Aplicação django-bootstrap3

Usaremos o django-bootstrap3 para integrar o Bootstrap em nosso projeto. Essa aplicação faz download dos arquivos necessários do Bootstrap, coloca-os em um lugar apropriado em seu projeto e deixa as diretivas de estilização disponíveis nos templates de seu projeto.

Para instalar o django-bootstrap3 execute o comando a seguir em um ambiente virtual ativo:

```
(ll_env)learning_log$ pip install django-bootstrap3
--trecho omitido--
Successfully installed django-bootstrap3
```

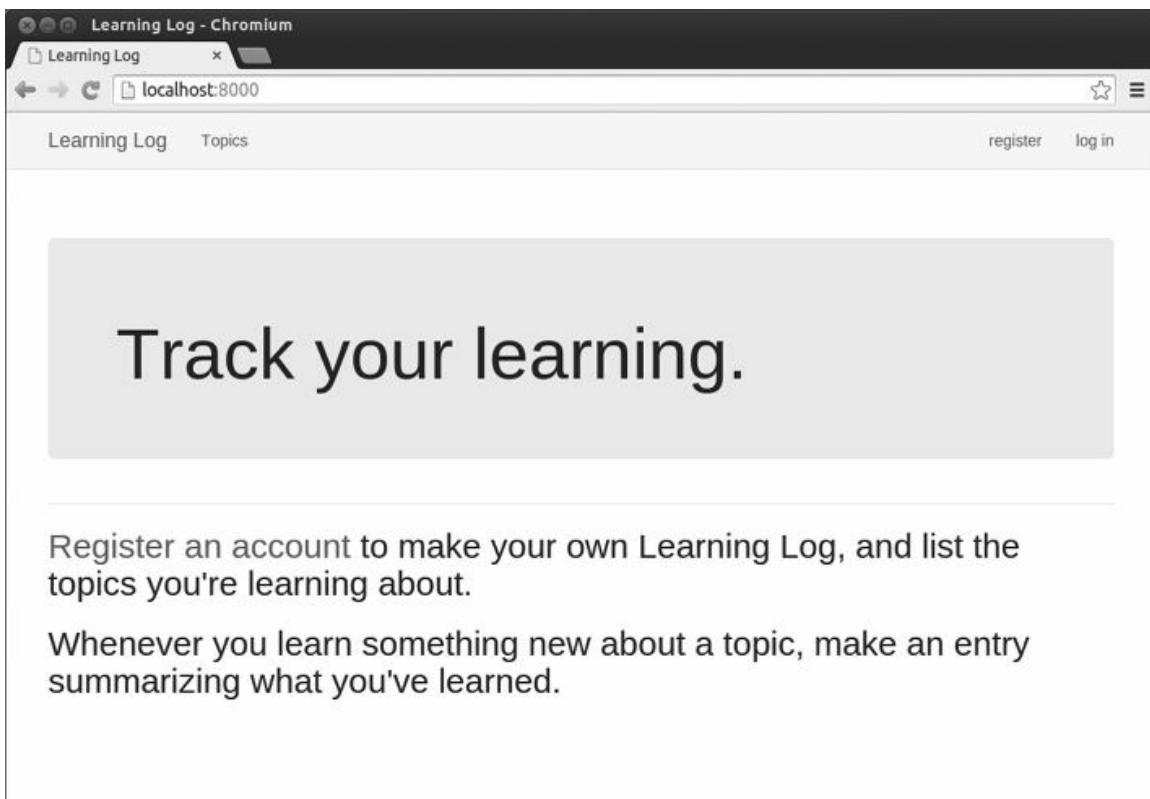
Na sequência adicione o código a seguir a fim de incluir o django-boostrap3 em `INSTALLED_APPS` no arquivo *settings.py*:

settings.py

```
--trecho omitido--
INSTALLED_APPS = (
    --trecho omitido--
    'django.contrib.staticfiles',

    # Aplicações de terceiros
    'bootstrap3',

    # Minhas aplicações
    'learning_logs',
    'users',
)
--trecho omitido--
```

Inicie uma nova seção, chamada *Aplicações de terceiros*, para aplicações criadas por outros desenvolvedores e adicione `'bootstrap3'` nesta seção. A maioria das aplicações deve ser incluída em `INSTALLED_APPS`, mas para ter certeza, leia as instruções de configuração da aplicação que você estiver usando.

Precisamos que django-bootstrap3 inclua a jQuery: uma biblioteca JavaScript que possibilita o uso de alguns dos elementos interativos oferecidos pelos templates do Bootstrap. Adicione o código a seguir no final de *settings.py*:

settings.py

```
--trecho omitido--
# Minhas configurações
LOGIN_URL = '/users/login/'

# Configurações para django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }
```

Esse código evita que tenhamos de fazer download da jQuery e colocá-la no lugar correto manualmente.

Usando o Bootstrap para estilizar Learning Log

O Bootstrap é basicamente uma grande coleção de ferramentas de estilização. Ele também tem diversos templates que podem ser aplicados em seu projeto para criar um estilo de modo geral. Se você está apenas começando, será muito mais fácil usar esses templates que usar ferramentas individuais de estilização. Para ver os templates oferecidos pelo Bootstrap, consulte a seção *Getting Started* (Introdução) em *http://getbootstrap.com/*; então faça rolagens para baixo até o título *Examples* (Exemplos) e procure a seção *Navbars in action* (Barras de navegação em ação). Usaremos o template *Static top navbar*, que oferece uma barra de navegação superior simples, um cabeçalho de página e um contêiner para o conteúdo da página.

A Figura 20.1 mostra como será a aparência da página inicial depois que aplicarmos o template do Bootstrap em *base.html* e modificarmos um pouco o arquivo *index.html*.

Figura 20.1 – A página inicial de Learning Log usando o Bootstrap.

Agora que você já sabe qual é o resultado que estamos buscando, as próximas seções serão mais fáceis de entender.

Modificando base.html

Precisamos modificar o template *base.html* para acomodar o template do Bootstrap. Apresentarei o novo *base.html* por partes.

Definindo os cabeçalhos HTML

A primeira mudança em *base.html* define os cabeçalhos HTML no arquivo, de modo que sempre que uma página de Learning Log for aberta, a barra de título do navegador exibirá o nome do site. Também acrescentaremos alguns requisitos para usar o Bootstrap em nossos templates. Apague tudo que está em *base.html* e substitua pelo código a seguir:

base.html

```
u {% load bootstrap3 %}
```

```
v <!DOCTYPE html>
w <html lang="en">
x   <head>
      <meta charset="utf-8">
      <meta http-equiv="X-UA-Compatible" content="IE=edge">
      <meta name="viewport" content="width=device-width, initial-
  scale=1">

y     <title>Learning Log</title>

z     {% bootstrap_css %}
      {% bootstrap_javascript %}

{   </head>
```

Em u carregamos a coleção de tags de template disponíveis em django-bootstrap3. Em seguida declaramos esse arquivo como um documento HTML v escrito em inglês w. Um arquivo HTML é dividido em duas partes principais, o *cabeçalho* e o *corpo* – o cabeçalho do arquivo começa em x. O cabeçalho de um arquivo HTML não tem nenhum conteúdo: ele simplesmente informa o que o navegador deve saber para exibir a página de forma correta. Em y incluímos um elemento `title` para a página, que será exibido na barra de título do navegador sempre que Learning Log for aberto.

Em z usamos uma das tags de template personalizadas de django-bootstrap3, que diz a Django para incluir todos os arquivos de estilização do Bootstrap. A tag seguinte habilita todo o comportamento interativo que possa ser usado em uma página, por exemplo, barras de navegação possíveis de serem contraídas. Em { temos a tag de fechamento `</head>`.

Definindo a barra de navegação

Agora definiremos a barra de navegação na parte superior da página:

```
--trecho omitido--
  </head>

  <body>
```

```
      <!-- Static navbar -->
u     <nav class="navbar navbar-default navbar-static-top">
        <div class="container">

          <div class="navbar-header">
v           <button type="button" class="navbar-toggle collapsed"
                data-toggle="collapse" data-target="#navbar"
                aria-expanded="false" aria-controls="navbar">
            </button>
w           <a class="navbar-brand" href="{% url 'learning_logs:index'
%}">
                Learning Log</a>
          </div>

x         <div id="navbar" class="navbar-collapse collapse">
y           <ul class="nav navbar-nav">
z             <li><a href="{% url 'learning_logs:topics' %}">Topics</a>
</li>
            </ul>

{           <ul class="nav navbar-nav navbar-right">
              {% if user.is_authenticated %}
                <li><a>Hello, {{ user.username }}.</a></li>
                <li><a href="{% url 'users:logout' %}">log out</a></li>
              {% else %}
                <li><a href="{% url 'users:register' %}">register</a>
  </li>
                <li><a href="{% url 'users:login' %}">log in</a></li>
              {% endif %}
|           </ul>
          </div><!--/.nav-collapse -->

        </div>
      </nav>
```

O primeiro elemento é a tag de abertura `<body>`. O corpo de um arquivo HTML contém o que os usuários verão em uma página. Em u temos um elemento `<nav>` que representa a seção de links de navegação da página. Tudo que estiver nesse elemento será estilizado de acordo com as regras de estilo do Bootstrap definidas pelos seletores `navbar`, `navbar-default` e

navbar-static-top. Um *seletor* determina a quais elementos de uma página uma determinada regra de estilo se aplica.

Em v o template define um botão que aparecerá se a janela do navegador for estreita demais para exibir toda a barra de navegação horizontalmente. Quando o usuário clicar no botão, os elementos de navegação aparecerão em uma lista suspensa. A referência collapse faz a barra de navegação ser contraída quando o usuário reduzir a janela do navegador ou quando o site for exibido em dispositivos móveis, com telas menores.

Em w configuramos o nome do projeto para que apareça na extremidade esquerda da barra de navegação e fizemos dele um link para a página inicial, pois essa informação aparecerá em todas as páginas do projeto.

Em x definimos um conjunto de links que permite aos usuários navegar pelo site. Uma barra de navegação é basicamente uma lista que começa com <ul> y, e cada link é um item dessa lista (<li>) z. Para acrescentar mais links, insira outras linhas usando a estrutura a seguir:

```
<li><a href="{% url 'learning_logs:título' %}">Título</a></li>
```

Essa linha representa um único link na barra de navegação. O link foi obtido diretamente da versão anterior de *base.html*.

Em { colocamos uma segunda lista de links de navegação, dessa vez usando o seletor navbar-right. Esse seletor estiliza o conjunto de links para que ele apareça na borda direita da barra de navegação, em um local em que você normalmente vê links para login e para cadastramento. Nesse caso, exibiremos a saudação ao usuário e o link para logout ou os links para cadastramento ou login. O restante do código dessa seção fecha os elementos que contêm a barra de navegação |.

Definindo a parte principal da página

O restante de *base.html* contém a parte principal da página:

```
  --trecho omitido--
    </nav>

u   <div class="container">
```

```
        <div class="page-header">
v         {% block header %}{% endblock header %}
        </div>
        <div>
w         {% block content %}{% endblock content %}
        </div>

      </div> <!-- /container -->

    </body>
  </html>
```

Em u temos uma div de abertura com a classe `container`. Uma *div* é uma seção de uma página web que pode ser usada para qualquer finalidade, e pode ser estilizada com uma borda, espaços em torno do elemento (margens), espaço entre o conteúdo e a borda (preenchimento), cores de fundo e outras regras de estilo. Essa div em particular atua como um contêiner no qual colocamos dois elementos: um novo bloco chamado `header` v e o bloco `content`, que usamos no Capítulo 18 w. O bloco de cabeçalho diz aos usuários quais tipos de informação estão na página e o que eles podem fazer. Esse bloco tem a classe `page-header`, que aplica um conjunto de regras de estilo ao bloco. O bloco de conteúdo está em uma div separada, sem uma classe específica de estilo.

Ao carregar a página inicial de Learning Log em um navegador, você deverá ver uma barra de navegação de aspecto profissional, semelhante àquela que vimos na Figura 20.1. Experimente redimensionar a janela para que ela fique realmente estreita; a barra de navegação deverá ser substituída por um botão. Clique no botão; todos os links deverão aparecer em uma lista suspensa.

NOTA Essa versão simplificada do template do Bootstrap deve funcionar na maioria dos navegadores recentes. Navegadores mais antigos talvez não renderizem alguns estilos corretamente. O template completo, que está em *http://getbootstrap.com/getting-started/#examples/*, funcionará em quase todos os navegadores disponíveis.

Estilizando a página inicial usando um jumbotron

Vamos atualizar a página inicial usando o bloco `header` recém-definido e

outro elemento do Bootstrap chamado *jumbotron* – uma caixa grande que se destacará do restante da página e pode conter o que você quiser. Geralmente ele é usado em páginas iniciais para uma rápida descrição do projeto como um todo. Vamos aproveitar e atualizar a mensagem na página inicial também. Eis o código em *index.html*:

index.html

```
{% extends "learning_logs/base.html" %}

u {% block header %}
v   <div class='jumbotron'>
      <h1>Track your learning.</h1>
    </div>
  {% endblock header %}

  {% block content %}
w   <h2>
      <a href="{% url 'users:register' %}">Register an account</a> to
  make
      your own Learning Log, and list the topics you're learning about.
    </h2>
    <h2>
      Whenever you learn something new about a topic, make an entry
      summarizing what you've learned.
    </h2>
  {% endblock content %}
```

Em u dizemos a Django que estamos prestes a definir o que estará no bloco `header`. Em um elemento `jumbotron` v, colocamos uma tagline concisa, *Track your learning*, para dar uma noção do que o Learning Log faz àqueles que visitarem o site pela primeira vez.

Em w acrescentamos um texto que oferece mais algumas informações. Convidamos as pessoas a criar uma conta e descrevemos as duas principais ações: adicionar novos assuntos e criar entradas para um assunto. A página de índice agora tem a aparência mostrada na Figura 20.1 e representa uma melhoria significativa em relação ao nosso projeto sem estilização.

Estilizando a página de login

Melhoramos o aspecto geral da página de login, mas não o formulário de login, portanto vamos deixar o formulário consistente com o restante da página:

login.html

```
  {% extends "learning_logs/base.html" %}
u {% load bootstrap3 %}

v {% block header %}
    <h2>Log in to your account.</h2>
  {% endblock header %}

  {% block content %}

w   <form method="post" action="{% url 'users:login' %}" class="form">
      {% csrf_token %}
x     {% bootstrap_form form %}

y     {% buttons %}
        <button name="submit" class="btn btn-primary">log in</button>
      {% endbuttons %}

      <input type="hidden" name="next" value="{% url
  'learning_logs:index' %}" />
    </form>

  {% endblock content %}
```

Em u carregamos as tags de template do bootstrap3 nesse template. Em v definimos o bloco `header`, que descreve para que serve a página. Observe que removemos o bloco `{% if form.errors %}` do template; o django-bootstrap3 administra erros de formulário automaticamente.

Em w adicionamos um atributo `class="form"` e então usamos a tag de template `{% bootstrap_form %}` quando exibimos o formulário x; ela substitui a tag `{{ form.as_p }}` que usamos no Capítulo 19. A tag de template `{% booststrap_form %}` insere as regras de estilo do Bootstrap nos elementos individuais do formulário quando esses são renderizados. Em y abrimos uma tag de template `{% buttons %}` do bootstrap3, que adiciona a

estilização do Bootstrap aos botões.

A Figura 20.2 mostra o formulário de login conforme renderizado agora.

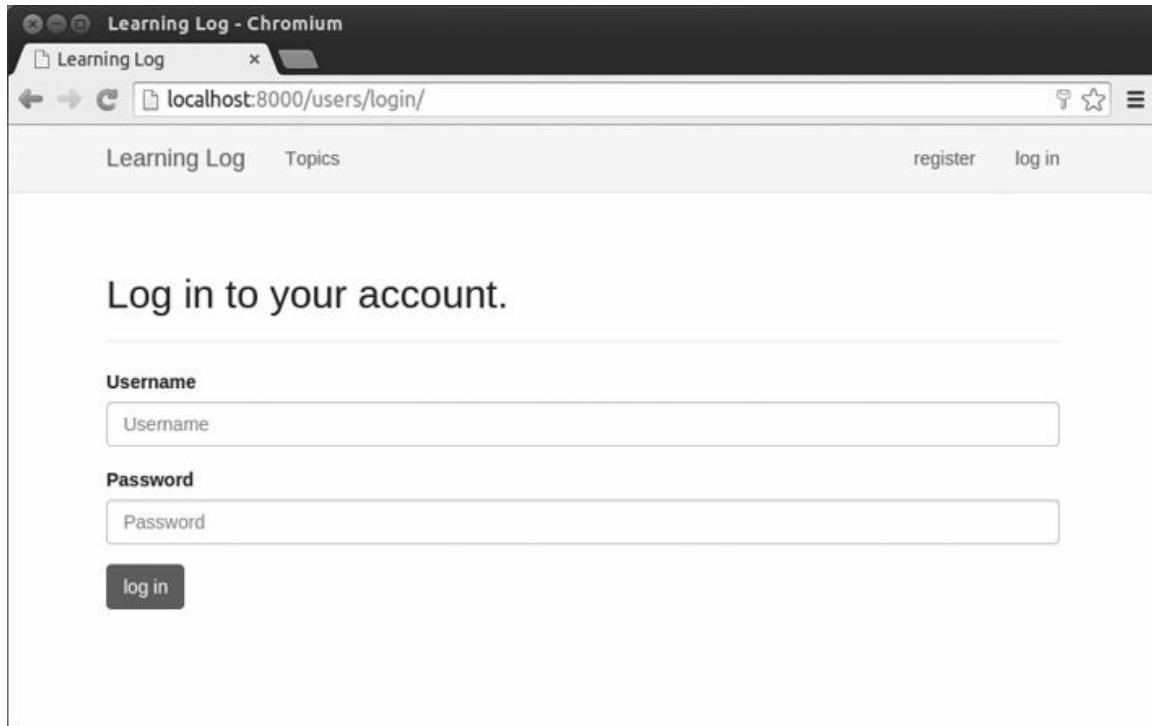



Figura 20.2 – A página de login estilizada com o Bootstrap.

A página está bem mais limpa, além de ter uma estilização consistente e um propósito claro. Experimente fazer login com um nome de usuário ou uma senha incorretos; você verá que até mesmo as mensagens de erro estão estilizadas de forma consistente e se integram bem ao site como um todo.

Estilizando a página new_topic

Vamos fazer o restante das páginas serem consistentes também. Atualizaremos a página `new_topic` a seguir:

new_topic.html

```
  {% extends "learning_logs/base.html" %}
  {% load bootstrap3 %}

u {% block header %}
    <h2>Add a new topic:</h2>
  {% endblock header %}
```

```
  {% block content %}

v   <form action="{% url 'learning_logs:new_topic' %}" method='post'
        class="form">

      {% csrf_token %}
w     {% bootstrap_form form %}

x     {% buttons %}
        <button name="submit" class="btn btn-primary">add topic</button>
      {% endbuttons %}

    </form>

  {% endblock content %}
```

A maior parte das alterações, nesse caso, são semelhantes àquelas aplicadas em *login.html*: carregamos bootstrap3 e adicionamos o bloco `header` com uma mensagem apropriada em u. Então acrescentamos o atributo `class="form"` na tag `<form>` v, usamos a tag de template `{% bootstrap_form %}` no lugar de `{{ form.as_p }}` w e utilizamos a estrutura do bootstrap3 para o botão de submissão x. Faça login e navegue para a página `new_topic`; ela deverá estar parecida com a página de login agora.

Estilizando a página de assuntos

Vamos agora garantir que as páginas para visualizar informações estejam apropriadamente estilizadas também, começando pela página de assuntos:

topics.html

```
  {% extends "learning_logs/base.html" %}

u {% block header %}
    <h1>Topics</h1>
  {% endblock header %}

  {% block content %}
```

```
    <ul>
      {% for topic in topics %}
        <li>
v           <h3>
              <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic
  }}</a>
          </h3>
        </li>
      {% empty %}
        <li>No topics have been added yet.</li>
      {% endfor %}
    </ul>

w   <h3><a href="{% url 'learning_logs:new_topic' %}">Add new topic</h3>

  {% endblock content %}
```

Não precisamos da tag `{% load bootstrap3 %}`, pois não estamos usando nenhuma tag de template personalizada do bootstrap3 nesse arquivo. Adicionamos o cabeçalho *Topics* no bloco `header` u. Estilizamos cada assunto como um elemento `<h3>` para deixá-los um pouco maiores na página v e fizemos o mesmo com o link para acrescentar um novo assunto w.

Estilizando as entradas na página de um assunto

A página de um assunto tem mais conteúdo que a maioria das páginas, portanto exigirá um pouco mais de trabalho. Usaremos os painéis do Bootstrap para dar destaque a cada entrada. Um *painel* é uma div com estilização predefinida, e é perfeito para exibir as entradas de um assunto:

topic.html

```
  {% extends 'learning_logs/base.html' %}

u {% block header %}
    <h2>{{ topic }}</h2>
  {% endblock header %}

  {% block content %}
```

```
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">add new
entry</a>
  </p>

  {% for entry in entries %}
v   <div class="panel panel-default">
w     <div class="panel-heading">
x       <h3>
          {{ entry.date_added|date:'M d, Y H:i' }}
y         <small>
            <a href="{% url 'learning_logs:edit_entry' entry.id %}">
              edit entry</a>
          </small>
        </h3>
      </div>
z     <div class="panel-body">
        {{ entry.text|linebreaks }}
      </div>
    </div> <!-- panel -->
  {% empty %}
    There are no entries for this topic yet.
  {% endfor %}

{% endblock content %}
```

Inicialmente colocamos o assunto no bloco `header` u. Então apagamos a estrutura de lista não ordenada usada antes nesse template. Em vez de fazer com que cada entrada seja um item da lista, criamos um elemento div `panel` em v, que contém duas outras divs aninhadas: uma div `panel-heading` w e uma div `panel-body` z. A div `panel-heading` contém a data da entrada e o link para editá-la. Ambos são estilizados como elementos `<h3>` x, mas adicionamos tags `<small>` em torno do link `edit_entry` para deixá-lo um pouco menor que o timestamp y.

Em z temos a div `panel-body`, que contém o texto propriamente dito da entrada. Observe que o código de Django para incluir a informação na página não mudou; somente os elementos que afetam a aparência da página foram alterados.

A Figura 20.3 mostra a página de um assunto com sua nova aparência. As funcionalidades de Learning Log não mudaram, mas ele parece mais profissional e convidativo aos usuários.

Figura 20.3 – A página de um assunto com estilização do Bootstrap.

NOTA Se quiser usar um template diferente do Bootstrap, siga um processo semelhante ao que usamos até agora neste capítulo. Copie o template para *base.html* e modifique os elementos com conteúdos para que o template exiba as informações de seu projeto. Então use as ferramentas individuais de estilização do Bootstrap para estilizar o conteúdo de cada página.

FAÇA VOCÊ MESMO

20.1 – Outros formulários: Aplicamos os estilos do Bootstrap nas páginas `login` e `add_topic`. Faça mudanças semelhantes no restante das páginas baseadas em formulários: `new_entry`, `edit_entry` e `register`.

20.2 – Blog com estilo: Use o Bootstrap para estilizar o projeto Blog que você criou no Capítulo 19.

Implantação do projeto Learning Log

Agora que temos um projeto de aspecto profissional, vamos implantá-lo em um servidor ativo para que qualquer pessoa com uma conexão de internet possa usá-lo. Utilizaremos o Heroku: uma plataforma baseada em web que permite administrar a implantação de aplicações web. Deixaremos o Learning Log pronto para executar no Heroku.

O processo é um pouco diferente no Windows em comparação com o Linux e o OS X. Se você usa Windows, verifique as observações em cada seção que especificam o que você deverá fazer de modo diferente em seu sistema.

Criando uma conta no Heroku

Para criar uma conta acesse *https://heroku.com/* e clique em um dos links para inscrição. Não é necessário pagar para criar uma conta e o Heroku tem um nível gratuito que permite testar seus projetos em uma implantação ativa.

NOTA O nível gratuito do Heroku tem limites, por exemplo, quanto ao número de aplicações que você pode implantar e a frequência com que as pessoas podem visitar sua aplicação. Porém, esses limites são generosos o suficiente para permitir que você exercite a implantação de aplicações sem qualquer custo.

Instalando o Heroku Toolbelt

Para implantar e administrar um projeto nos servidores do Heroku, você precisará das ferramentas disponíveis no Heroku Toolbelt. Para instalar a versão mais recente do Heroku Toolbelt, acesse *https://toolbelt.heroku.com/* e siga as instruções para o seu sistema operacional, que incluirão um comando de uma linha no terminal ou um instalador, que você poderá baixar e executar.

Instalando os pacotes necessários

Você também precisará instalar alguns pacotes que ajudam a servir projetos Django em um servidor ativo. Em um ambiente virtual ativo, execute os comandos a seguir:

```
(ll_env)learning_log$ pip install dj-database-url
(ll_env)learning_log$ pip install dj-static
(ll_env)learning_log$ pip install static3
(ll_env)learning_log$ pip install gunicorn
```

Execute os comandos um de cada vez para saber se houve falha na instalação de algum pacote. O pacote `dj-database-url` ajuda o Django a se comunicar com o banco de dados usado pelo Heroku, `dj-static` e `static3` ajudam o Django a administrar arquivos estáticos corretamente e o gunicorn é um servidor capaz de servir aplicações em um ambiente ativo. (*Arquivos estáticos* contêm regras de estilo e arquivos JavaScript.)

NOTA Alguns dos pacotes necessários talvez não sejam instalados no Windows, portanto não se preocupe se vir uma mensagem de erro quando tentar instalar alguns deles. O que importa é conseguir que o Heroku instale os pacotes na implantação ativa, e é isso que faremos na próxima seção.

Criando uma lista de pacotes com um arquivo requirements.txt

O Heroku precisa saber de quais pacotes o nosso projeto depende, portanto usaremos o pip para gerar um arquivo que os liste. Novamente, a partir de um ambiente virtual ativo execute o comando a seguir:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

O comando `freeze` diz ao pip para escrever os nomes de todos os pacotes instalados no momento para o projeto no arquivo *requirements.txt*. Abra esse arquivo para ver os pacotes e os números das versões instaladas em seu projeto (usuários de Windows talvez não vejam todas essas linhas):

requirements.txt

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
gunicorn==19.3.0
static3==0.6.1
```

O projeto Learning Log já depende de seis pacotes diferentes com números de versão específicos, portanto ele exige um ambiente particular

para executar de forma apropriada. Quando fizermos a implantação de Learning Log, o Heroku instalará todos os pacotes listados em *requirements.txt*, criando um ambiente com os mesmos pacotes utilizados localmente. Por esse motivo, podemos estar confiantes de que o projeto implantado se comportará do mesmo modo que em nosso sistema local. Essa será uma grande vantagem quando você começar a criar e a manter diversos projetos em seu sistema.

Em seguida, precisamos adicionar `psycopg2`, que ajuda o Heroku a administrar o banco de dados ativo, à lista de pacotes. Abra o arquivo *requirements.txt* e adicione a linha `psycopg2>=2.6.1`. Isso fará a versão 2.6.1 de `psycopg2` ser instalada, ou uma versão mais recente, se estiver disponível:

requirements.txt

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
gunicorn==19.3.0
static3==0.6.1
psycopg2>=2.6.1
```

Mesmo que algum dos pacotes não tenha sido instalado em seu sistema, acrescente-o também. Quando terminar, seu arquivo *requirements.txt* deverá incluir cada um dos pacotes mostrados anteriormente. Se um pacote estiver listado em seu sistema, mas o número da versão for diferente daquele mostrado aqui, mantenha a versão que você tem em seu sistema.

NOTA Se você usa Windows, garanta que sua versão de *requirements.txt* esteja de acordo com a lista mostrada aqui, independentemente de quais pacotes você conseguiu instalar em seu sistema.

Especificando o runtime de Python

A menos que você especifique uma versão de Python, o Heroku usará sua própria versão default de Python. Vamos garantir que ele use a mesma versão de Python que estamos usando. Em um ambiente virtual ativo, execute o comando `python --version`:

```
(ll_env)learning_log$ python --version
Python 3.5.0
```

Nesse exemplo estou executando Python 3.5.0. Crie um novo arquivo chamado *runtime.txt* no mesmo diretório em que está *manage.py* e forneça o seguinte:

runtime.txt

```
python-3.5.0
```

Esse arquivo deve conter uma linha com a sua versão de Python especificada no formato mostrado; lembre-se de usar `python` com letras minúsculas, seguido de um hífen e depois de um número de versão com três partes.

NOTA Se você obtiver um erro informando que o runtime de Python solicitado não está disponível, acesse *https://devcenter.heroku.com/* e clique em **Python**; em seguida, procure um link para *Specifying a Python Runtime* (Especificando um runtime de Python). Dê uma olhada no artigo para encontrar os runtimes disponíveis e utilize um que seja mais próximo à sua versão de Python.

Modificando settings.py para o Heroku

Agora precisamos acrescentar uma seção no final de *settings.py* a fim de definir algumas configurações específicas para o ambiente do Heroku:

settings.py

```
--trecho omitido--
# Configurações para django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }

# Configurações para o Heroku
u if os.getcwd() == '/app':
v     import dj_database_url
      DATABASES = {
          'default':
dj_database_url.config(default='postgres://localhost')
    }
```

```
    # Honra o cabeçalho 'X-Forwarded-Proto' para request.is_secure()
w   SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

    # Cabeçalhos para permitir todos os hosts
x   ALLOWED_HOSTS = ['*']

    # Configuração de recursos estáticos
y   BASE_DIR = os.path.dirname(os.path.abspath(__file__))
    STATIC_ROOT = 'staticfiles'
    STATICFILES_DIRS = (
        os.path.join(BASE_DIR, 'static'),
    )
```

Em u usamos a função `getcwd()`, que obtém o *diretório de trabalho atual* em que o arquivo está executando. Em uma implantação no Heroku, o diretório é sempre */app*. Em uma implantação local, o diretório normalmente é o nome da pasta do projeto (*learning_log* em nosso caso). O teste `if` garante que as configurações desse bloco se apliquem somente quando o projeto estiver implantado no Heroku. Essa estrutura possibilita ter um único arquivo de configuração que funcione em nosso ambiente de desenvolvimento local bem como no servidor ativo.

Em v importamos `dj_database_url` para ajudar a configurar o banco de dados no Heroku. O Heroku utiliza o PostgreSQL (também chamado de Postgres) – um banco de dados mais sofisticado que o SQLite – e esses parâmetros configuram o projeto para usar o Postgres no Heroku. O restante das configurações dá suporte às requisições HTTPS w, garante que o Django servirá o projeto a partir do URL do Heroku x e configura o projeto para servir arquivos estáticos corretamente no Heroku y.

Criando um Procfile para iniciar processos

Um *Procfile diz ao* Heroku quais processos devem ser iniciados para servir o projeto de forma apropriada. É um arquivo de uma só linha que você deve salvar como *Procfile*, com um *P* maiúsculo e sem extensão, no mesmo diretório em que está *manage.py*.

Eis a linha que está em *Procfile*:

Procfile

```
web: gunicorn learning_log.wsgi --log-file -
```

Essa linha diz ao Heroku para usar o gunicorn como servidor e utilizar as configurações em *learning_log/wsgi.py* para iniciar a aplicação. A flag `log-file` diz ao Heroku quais são os tipos de eventos que devem ser registrados no log.

Modificando wsgi.py para o Heroku

Também é necessário modificar o arquivo *wsgi.py* para o Heroku, pois ele precisa de uma configuração um pouco diferente daquela que estávamos usando:

wsgi.py

```
--trecho omitido--
import os

from django.core.wsgi import get_wsgi_application
from dj_static import Cling

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
"learning_log.settings")
application = Cling(get_wsgi_application())
```

Importamos o Cling, que ajuda a servir arquivos estáticos de forma correta, e o usamos para iniciar a aplicação. Esse código funcionará localmente também, portanto não é necessário colocá-lo em um bloco `if`.

Criando um diretório para arquivos estáticos

No Heroku, o Django reúne todos os arquivos estáticos e os coloca em um só lugar para que possam ser administrados de modo eficiente. Criaremos um diretório para esses arquivos estáticos. Na pasta *learning_log* em que estivemos trabalhando, há outra pasta chamada *learning_log*. Nessa pasta interna, crie uma nova pasta de nome *static*, com o path *learning_log/learning_log/static/*. Também precisamos criar um arquivo placeholder a ser armazenado nesse diretório por enquanto, pois diretórios

vazios não serão incluídos no projeto quando ele for enviado ao Heroku. No diretório *static/*, crie um arquivo chamado *placeholder.txt*:

placeholder.txt

```
Este arquivo garante que learning_log/static/ será adicionado ao
projeto.
O Django reunirá os arquivos estáticos e os colocará em
learning_log/static/.
```

Não há nada de especial nesse texto; ele simplesmente nos lembra por que incluímos esse arquivo no projeto.

Usando o servidor gunicorn localmente

Se você usa Linux ou OS X, poderá tentar utilizar o servidor gunicorn localmente antes de fazer a implantação no Heroku. A partir de um ambiente virtual ativo, execute o comando `heroku local` para iniciar os processos definidos em *Procfile*:

```
  (ll_env)learning_log$ heroku local
  Installing Heroku Toolbelt v4... done
  --trecho omitido--
  forego | starting web.1 on port 5000
u web.1  | [2015-08-13 22:00:45 -0800] [12875] [INFO] Starting gunicorn
19.3.0
v web.1  | [2015-08-13 22:00:45 -0800] [12875] [INFO] Listening at:
     http://0.0.0.0:5000 (12875)
w web.1  | [2015-08-13 22:00:45 -0800] [12878] [INFO] Booting worker with
pid: 12878
```

Na primeira vez que você executar `heroku local` diversos pacotes do Heroku Toolbelt serão instalados. A saída mostra que o gunicorn foi iniciado com um id de processo igual a 12875 nesse exemplo u. Em v o gunicorn está ouvindo requisições na porta 5000. Além disso, o gunicorn iniciou um processo de *trabalho* – ou *worker* – (12878) para ajudá-lo a atender às requisições w.

Acesse *http://localhost:5000/* para garantir que tudo esteja funcionando; você deverá visualizar a página inicial de Learning Log exatamente como ela aparece quando usamos o servidor Django (`runserver`). Tecle CTRL-C

para interromper os processos iniciados por `heroku local`. Continue usando `runserver` no desenvolvimento local.

NOTA O gunicorn não executará no Windows, portanto ignore esse passo se você usa esse sistema operacional. Isso não afetará a sua capacidade de fazer a implantação do projeto no Heroku.

Usando o Git para monitorar os arquivos do projeto

Se você leu o Capítulo 17, saberá que o Git é um programa de controle de versões que permite salvar um snapshot (imagem instantânea) do código de seu projeto sempre que uma nova funcionalidade for implementada com sucesso. Isso permite retornar facilmente ao último snapshot de seu projeto que esteja funcionando caso algo dê errado, por exemplo, se você introduzir um bug por acidente enquanto trabalha em uma nova funcionalidade. Cada um desses snapshots é denominado *commit*.

Usar o Git significa que você pode tentar implementar novas funcionalidades sem se preocupar em causar falhas em seu projeto. Quando fizer a implantação em um servidor ativo, será necessário garantir que você esteja implantando uma versão de seu projeto que funcione. Se quiser ler mais sobre o Git e sobre controle de versões, consulte o Apêndice D.

Instalando o Git

O Heroku Toolbelt inclui o Git, portanto ele já deverá estar instalado em seu sistema. No entanto, janelas de terminal abertas antes da instalação do Heroku Toolbelt não terão acesso ao Git, assim, você deve abrir uma nova janela de terminal e executar o comando `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.5.0
```

Se obtiver uma mensagem de erro por algum motivo, veja as instruções para instalação do Git no Apêndice D.

Configurando o Git

O Git controla quem fez as alterações em um projeto, mesmo em casos

como este, em que há apenas uma pessoa trabalhando nele. Para isso o Git precisa ter o seu nome de usuário e o seu email. Você deve fornecer um nome de usuário, mas sinta-se à vontade para criar um email para os projetos que servirão como exercício:

```
(ll_env)learning_log$ git config --global user.name "ehmatthes"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

Se você se esquecer desse passo, o Git solicitará essas informações quando você fizer o seu primeiro commit.

Ignorando arquivos

Não é necessário que o Git monitore todos os arquivos do projeto, portanto diremos a ele para ignorar alguns. Crie um arquivo chamado *.gitignore* na pasta que contém *manage.py*. Observe que o nome desse arquivo começa com um ponto e não há nenhuma extensão. Eis o conteúdo de *.gitignore*:

.gitignore

```
ll_env/
__pycache__/
*.sqlite3
```

Dissemos ao Git para ignorar todo o diretório *ll_env*, pois ele pode ser recriado automaticamente a qualquer momento. Também não monitoramos o diretório *__pycache__*, que contém os arquivos *.pyc*, criados de modo automático quando Django executa os arquivos *.py*. Não monitoramos as alterações no banco de dados local, pois é um hábito ruim: se você algum dia usar o SQLite em um servidor, poderá acidentalmente sobrescrever o banco de dados ativo com o seu banco de dados local de testes quando enviar o projeto para o servidor.

NOTA Se você usa Python 2.7, substitua `__pycache__` por `*.pyc` porque o Python 2.7 não cria um diretório *__pycache__*.

Fazendo commit do projeto

Devemos inicializar um repositório no Git para Learning Log, acrescentar

todos os arquivos necessários no repositório e fazer commit do estado inicial do projeto. Fazemos isso assim:

```
u (ll_env)learning_log$ git init
  Initialized empty Git repository in
  /home/ehmatthes/pcc/learning_log/.git/
v (ll_env)learning_log$ git add .
w (ll_env)learning_log$ git commit -am "Ready for deployment to heroku."
  [master (root-commit) dbc1d99] Ready for deployment to heroku.
  43 files changed, 746 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 Procfile
  --trecho omitido--
  create mode 100644 users/views.py
x (ll_env)learning_log$ git status
  # On branch master
  nothing to commit, working directory clean
  (ll_env)learning_log$
```

Em u executamos o comando `git init` para inicializar um repositório vazio no diretório que contém Learning Log. Em v usamos o comando `git add .` para adicionar todos os arquivos que não serão ignorados no repositório. (Lembre-se do ponto.) Em w executamos o comando `git commit -am` *`mensagem de commit`*: a flag `-a` diz ao Git para incluir todos os arquivos alterados nesse commit, enquanto a flag `-m` lhe diz para registrar uma mensagem de log.

A execução do comando `git status` x informa que estamos no branch *master* e que o nosso diretório de trabalho está limpo (*clean*). Esse é o status que você vai querer ver sempre que enviar o seu projeto ao Heroku.

Enviado o projeto ao Heroku

Finalmente estamos prontos para enviar o projeto ao Heroku. Em uma sessão ativa de terminal, execute os comandos a seguir:

```
u (ll_env)learning_log$ heroku login
  Enter your Heroku credentials.
  Email: eric@example.com
  Password (typing will be hidden):
```

```
  Logged in as eric@example.com
v (ll_env)learning_log$ heroku create
  Creating afternoon-meadow-2775... done, stack is cedar-14
  https://afternoon-meadow-2775.herokuapp.com/ |
      https://git.heroku.com/afternoon-meadow-2775.git
  Git remote heroku added
w (ll_env)learning_log$ git push heroku master
  --trecho omitido--
  remote: -----> Launching... done, v6
x remote:        https://afternoon-meadow-2775.herokuapp.com/ deployed to
Heroku
  remote: Verifying deploy.... done.
  To https://git.heroku.com/afternoon-meadow-2775.git
     bdb2a35..62d711d  master -> master
  (ll_env)learning_log$
```

Em primeiro lugar, faça login no Heroku na sessão de terminal com o nome de usuário e a senha usados para criar uma conta em *https://heroku.com/* u. Em seguida diga ao Heroku para criar um projeto vazio v. O Heroku gera um nome composto de duas palavras e um número; você pode mudar isso depois. Então executamos o comando `git push heroku master` w, que diz ao Git para enviar o branch master do projeto para o repositório que o Heroku acabou de criar. O Heroku então constrói o projeto em seus servidores usando esses arquivos. Em x está o URL que usaremos para acessar o projeto ativo.

Ao executar esses comandos, o projeto será implantado, mas não estará totalmente configurado. Para conferir se o processo do servidor iniciou corretamente, utilize o comando `heroku ps`:

```
  (ll_env)learning_log$ heroku ps
u Free quota left: 17h 40m
v === web (Free): `gunicorn learning_log.wsgi __log-file -`
  web.1: up 2015/08/14 07:08:51 (~ 10m ago)
  (ll_env)learning_log$
```

A saída mostra por mais quanto tempo o projeto poderá estar ativo nas próximas 24 horas u. Na época em que este livro foi escrito, o Heroku permitia que as implantações gratuitas permanecessem ativas por até 18 horas em qualquer período de 24 horas. Se um projeto exceder esse limite,

uma página de erro padrão do servidor será exibida; vamos personalizar essa página de erro em breve. Em v vemos que o processo definido em *Procfile* foi iniciado.

Agora podemos abrir a aplicação em um navegador usando o comando `heroku open`:

```
(ll_env)learning_log$ heroku open
Opening afternoon-meadow-2775... done
```

Esse comando evita que você precise abrir um navegador e fornecer o URL que o Heroku mostrou, mas essa é outra maneira de acessar o site. Você deverá ver a página inicial de Learning Log, estilizada corretamente. No entanto, não será possível usar a aplicação ainda, pois não configuramos o banco de dados.

NOTA O processo de implantação do Heroku muda ocasionalmente. Se houver algum problema que você não tenha conseguido resolver, consulte a documentação do Heroku para obter ajuda. Acesse *https://devcenter.heroku.com/*, clique em **Python** e procure um link para *Getting Started with Django* (Introdução ao Django). Se não conseguir entender o que está aí, dê uma olhada nas sugestões do Apêndice C.

Configurando o banco de dados no Heroku

Precisamos executar `migrate` uma vez para configurar o banco de dados ativo e aplicar todas as migrações que geramos durante o desenvolvimento. Você pode executar comandos de Django e de Python em um projeto Heroku usando o comando `heroku run`. Eis o modo de executar `migrate` na implantação feita no Heroku:

```
u (ll_env)learning_log$ heroku run python manage.py migrate
v Running `python manage.py migrate` on afternoon-meadow-2775... up,
run.2435
    --trecho omitido--
w Running migrations:
    --trecho omitido--
    Applying learning_logs.0001_initial... OK
    Applying learning_logs.0002_entry... OK
    Applying learning_logs.0003_topic_user... OK
    Applying sessions.0001_initial... OK
```

```
(ll_env)learning_log$
```

Inicialmente executamos o comando `heroku run python manage.py migrate` u. O Heroku então cria uma sessão de terminal para executar o comando `migrate` v. Em w o Django aplica as migrações default e aquelas que geramos durante o desenvolvimento de Learning Log.

Agora, quando acessar a aplicação implantada, você deverá ser capaz de usá-la exatamente como foi feito em seu sistema local. Contudo, você não verá nenhum dos dados fornecidos em sua implantação local, pois não copiamos os dados para o servidor ativo. Essa é uma prática comum: normalmente não copiamos os dados locais para uma implantação ativa, pois os dados locais, em geral, são dados para testes.

Você pode compartilhar o seu link do Heroku para que qualquer pessoa possa usar a sua versão de Learning Log. Na próxima seção, concluiremos mais algumas tarefas para finalizar o processo de implantação e deixaremos você preparado para poder prosseguir com o desenvolvimento de Learning Log.

Aperfeiçoando a implantação no Heroku

Nesta seção aperfeiçoaremos a implantação criando um superusuário, como fizemos localmente. Também deixaremos o projeto mais seguro alterando a configuração `DEBUG` para `False`, de modo que os usuários não vejam nenhuma informação extra nas mensagens de erro, que poderiam ser usadas para atacar o servidor.

Criando um superusuário no Heroku

Já vimos que é possível executar comandos que devam ser aplicados uma só vez utilizando `heroku run`. Porém, você também pode executar comandos abrindo uma sessão de terminal Bash enquanto estiver conectado ao servidor Heroku usando o comando `heroku run bash`. *Bash* é a linguagem executada em muitos terminais Linux. Usaremos a sessão de terminal Bash a fim de criar um superusuário para que possamos ter acesso ao site de administração em uma aplicação ativa:

```
(ll_env)learning_log$ heroku run bash
```

```
  Running `bash` on afternoon-meadow-2775... up, run.6244
u ~ $ ls
  learning_log learning_logs manage.py Procfile requirements.txt
  runtime.txt
  users
  staticfiles
v ~ $ python manage.py createsuperuser
  Username (leave blank to use 'u41907'): ll_admin
  Email address:
  Password:
  Password (again):
  Superuser created successfully.
w ~ $ exit
  exit
  (ll_env)learning_log$
```

Em u executamos `ls` para ver quais arquivos e diretórios estão no servidor; devem ser os mesmos arquivos que temos em nosso sistema local. Você pode navegar por esse sistema de arquivos como faria com qualquer outro sistema.

NOTA Usuários de Windows usarão os mesmos comandos mostrados aqui (por exemplo, `ls` no lugar de `dir`), pois você estará executando um terminal Linux por meio de uma conexão remota.

Em v executamos o comando para criar um superusuário, cuja saída são os mesmos prompts que vimos em nosso sistema local quando criamos um superusuário no Capítulo 18. Quando terminar de criar o superusuário nessa sessão de terminal, utilize o comando `exit` para retornar à sessão de terminal de seu sistema local w.

Agora você pode adicionar */admin/* no final do URL da aplicação ativa e fazer login no site de administração. No meu caso, o URL é *https://afternoon-meadow-2775.herokuapp.com/admin/*.

Se outras pessoas já tiverem começado a usar o seu projeto, saiba que você terá acesso a todos os dados dessas pessoas! Não faça pouco caso desse fato; assim os usuários continuarão a confiar seus dados a você.

Criando um URL amigável ao usuário no Heroku

Provavelmente você vai querer que o URL seja mais amigável e mais fácil

de lembrar que *https://afternoon-meadow-2775.herokuapp.com/*. A aplicação pode ser renomeada com um único comando:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming afternoon-meadow-2775 to learning-log... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-
log.git
Git remote heroku updated
(ll_env)learning_log$
```

Você pode usar letras, números e traços ao nomear sua aplicação e lhe dar o nome que quiser, desde que ninguém mais tenha utilizado esse nome. Essa implantação agora está em *https://learning-log.herokuapp.com/*. O projeto não estará mais disponível no URL anterior; o comando `apps:rename` move totalmente o projeto para o novo URL.

NOTA Quando implantar o seu projeto usando o serviço gratuito do Heroku, ele colocará a sua implantação para dormir se ela não receber nenhuma requisição depois de determinado período de tempo ou se ela estiver ativa demais para o nível gratuito. Na primeira vez em que um usuário acessar o site depois que ele tiver dormido, demorará um pouco para que o site seja carregado, mas o servidor responderá às requisições subsequentes de modo mais rápido. É assim que o Heroku é capaz de oferecer implantações gratuitas.

Garantindo a segurança do projeto ativo

Há um problema de segurança evidente no modo como nosso projeto está implantado no momento: a configuração `DEBUG=True` em *settings.py*, que oferece mensagens de debug quando ocorrem erros. As páginas de erro de Django fornecem informações essenciais de debugging quando você está desenvolvendo um projeto, mas elas darão informações em demasia para os invasores se você as deixar habilitadas em um servidor ativo. Também devemos garantir que ninguém possa obter informações ou redirecionar requisições fingindo que é o host do projeto.

Vamos modificar *settings.py* para que seja possível ver as mensagens de erro localmente, mas não na implantação ativa.

settings.py

```
--trecho omitido--
```

```
  # Configurações para o Heroku
  if os.getcwd() == '/app':
      --trecho omitido--
      # Honra o cabeçalho 'X-Forwarded-Proto' para request.is_secure()
      SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

      # Permite que apenas o Heroku seja o host do projeto
u     ALLOWED_HOSTS = ['learning-log.herokuapp.com']

v     DEBUG = False

      # Configuração de recursos estáticos
      --trecho omitido--
```

Precisamos fazer apenas duas alterações: em u modificamos `ALLOWED_HOSTS` de modo que o único servidor que pode ser o host do projeto é o Heroku. Utilize o nome de sua aplicação, seja o nome fornecido pelo Heroku, como *afternoon-meadow-2775.herokuapp.com*, seja o nome que você escolheu. Em v definimos `DEBUG` com `False` para que o Django não compartilhe informações sensíveis quando um erro ocorrer.

Fazendo commit e enviando alterações

Agora precisamos fazer commit das alterações feitas em *settings.py* para o repositório do Git e, em seguida, enviá-las ao Heroku. Eis uma sessão de terminal que mostra esse processo:

```
u (ll_env)learning_log$ git commit -am "Set DEBUG=False for Heroku."
  [master 081f635] Set DEBUG=False for Heroku.
  1 file changed, 4 insertions(+), 2 deletions(-)
v (ll_env)learning_log$ git status
  # On branch master
  nothing to commit, working directory clean
  (ll_env)learning_log$
```

Executamos o comando `git commit` com uma mensagem de commit breve, porém descritiva u. Lembre-se de que a flag `-am` garante que o Git faça commit de todos os arquivos alterados e registre a mensagem de log. O Git reconhece que um arquivo mudou e faz commit dessa alteração no repositório.

Em `v` o status mostra que estamos trabalhando no branch master do repositório e que, no momento, não há nenhuma nova alteração que necessite de commit. É essencial verificar o status dessa mensagem antes de enviar o projeto ao Heroku. Se você não vir essa mensagem, o commit de algumas alterações não foi feito e essas mudanças não serão enviadas ao servidor. Você pode tentar executar o comando `commit` novamente, mas se não tiver certeza de como resolver o problema, leia o Apêndice D para entender melhor como trabalhar com o Git.

Agora vamos enviar o repositório atualizado ao Heroku:

```
(ll_env)learning_log$ git push heroku master
--trecho omitido--
remote: -----> Python app detected
remote: -----> Installing dependencies with pip
--trecho omitido--
remote: -----> Launching... done, v8
remote:        https://learning-log.herokuapp.com/ deployed to Heroku
remote: Verifying deploy.... done.
To https://git.heroku.com/learning-log.git
   4c9d111..ef65d2b  master -> master
(ll_env)learning_log$
```

O Heroku reconhece que o repositório foi alterado e reconstrói o projeto para garantir que todas as mudanças sejam levadas em consideração. Ele não reconstrói o banco de dados, portanto não será necessário executar `migrate` para essa atualização.

Para conferir se a implantação está mais segura agora, forneça o URL de seu projeto com uma extensão que não definimos. Por exemplo, tente acessar *http://learning-log.herokuapp.com/letmein/*. Você deverá ver uma página de erro genérica em sua implantação ativa, que não revelará nenhuma informação específica sobre o projeto. Se tentar fazer a mesma requisição na versão local de Learning Log em *http://localhost:8000/letmein/*, você deverá ver uma página de erro completa de Django. O resultado é perfeito: você verá mensagens de erro informativas quando estiver desenvolvendo o projeto, mas os usuários não verão informações críticas sobre o código do projeto.

Criando páginas de erro personalizadas

No Capítulo 19 configuramos Learning Log para que devolvesse um erro 404 caso o usuário requisitasse um assunto ou uma entrada que não lhe pertencesse. É provável que você já tenha visto alguns erros 500 de servidor (erros internos) a essa altura também. Um erro 404 geralmente significa que o seu código Django está correto, mas o objeto solicitado não existe; um erro 500 em geral quer dizer que há um erro no código que você escreveu, por exemplo, um erro em uma função de *views.py*. No momento, o Django devolve a mesma página de erro genérica nas duas situações, mas podemos escrever nossos próprios templates das páginas de erros 404 ou 500 que estejam de acordo com a aparência de Learning Log como um todo. Esses templates devem estar no diretório-raiz de templates.

Criando templates personalizados

Na pasta *learning_log/learning_log*, crie uma nova pasta chamada *templates*. Em seguida crie um novo arquivo de nome *404.html* contendo o código a seguir:

404.html

```
{% extends "learning_logs/base.html" %}

{% block header %}
    <h2>The item you requested is not available. (404)</h2>
{% endblock header %}
```

Esse template simples oferece informações para a página de erro 404 genérica, mas está estilizado para estar de acordo com o restante do site.

Crie outro arquivo de nome *500.html* contendo o código a seguir:

500.html

```
{% extends "learning_logs/base.html" %}

{% block header %}
    <h2>There has been an internal error. (500)</h2>
{% endblock header %}
```

Esses arquivos novos exigem uma pequena mudança em *settings.py*.

settings.py

```
--trecho omitido--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'learning_log/templates')],
        'APP_DIRS': True,
        --trecho omitido--
    },
]
--trecho omitido--
```

Essa alteração diz a Django para procurar os templates das páginas de erro no diretório-raiz de templates.

Visualizando as páginas de erro localmente

Se quiser ver a aparência das páginas de erro em seu sistema antes de enviá-las ao Heroku, inicialmente você deve definir `Debug=False` em suas configurações locais para suprimir as páginas de debug default de Django. Para isso, faça as seguintes alterações em *settings.py* (certifique-se de que está trabalhando na parte de *settings.py* que se aplica ao ambiente local, e não na parte que se aplica ao Heroku):

settings.py

```
--trecho omitido--
# AVISO DE SEGURANÇA: não execute com debug habilitado em produção!
DEBUG = False

ALLOWED_HOSTS = ['localhost']
--trecho omitido--
```

Você deve ter pelo menos um host especificado em `ALLOWED_HOSTS` quando `DEBUG` estiver definido com `False`. Agora solicite um assunto ou uma entrada que não pertença a você a fim de ver a página de erro 404, e requisite um URL que não exista (por exemplo, *localhost:8000/letmein/*) para ver a página de erro 500.

Quando terminar de conferir as páginas de erro, defina `DEBUG` novamente com `True` para continuar o desenvolvimento de Learning Log. (Garanta que `DEBUG` continue definido com `False` na seção de *settings.py* que se aplica à instalação no Heroku.)

NOTA A página de erro 500 não mostrará nenhuma informação sobre o usuário que está logado, pois o Django não envia nenhuma informação de contexto na resposta quando há um erro de servidor.

Enviando as alterações para o Heroku

Agora precisamos fazer commit das mudanças de template e enviá-las para que estejam ativas no Heroku:

```
u (ll_env)learning_log$ git add .
v (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error
pages."
  3 files changed, 15 insertions(+), 10 deletions(-)
  create mode 100644 learning_log/templates/404.html
  create mode 100644 learning_log/templates/500.html
w (ll_env)learning_log$ git push heroku master
  --trecho omitido--
  remote: Verifying deploy.... done.
  To https://git.heroku.com/learning-log.git
     2b34ca1..a64d8d3  master -> master
  (ll_env)learning_log$
```

Executamos o comando `git add .` em u porque criamos alguns arquivos novos no projeto, portanto é necessário dizer ao Git que comece a monitorar esses arquivos. Então fizemos commit das alterações v e enviamos o projeto atualizado ao Heroku w.

Agora, quando uma página de erro aparecer, ela deverá ter a mesma estilização que o restante do site, deixando a experiência do usuário mais uniforme quando surgirem erros.

Usando o método get_object_or_404()

A essa altura, se um usuário solicitar manualmente um assunto ou uma entrada que não exista, um erro de servidor 500 será informado. O Django tenta renderizar a página, mas não tem informações suficientes para isso, e

o resultado é um erro 500. Essa situação é mais bem tratada como um erro 404, e podemos implementar esse comportamento com a função de atalho `get_object_or_404()` de Django. Essa função tenta obter o objeto requisitado do banco de dados, mas se esse objeto não existir, uma exceção 404 será levantada. Importaremos essa função em *views.py* e a usaremos no lugar de `get()`:

views.py

```
--trecho omitido--
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect, Http404
--trecho omitido--
@login_required
def topic(request, topic_id):
    """Mostra um único assunto e todas as suas entradas."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Garante que o assunto pertence ao usuário atual
    --trecho omitido--
```

Agora, se solicitar um assunto que não exista (por exemplo, *http://localhost:8000/topics/999999/*), você verá uma página de erro 404. Para implantar essa alteração, faça um novo commit e então envie o projeto para o Heroku.

Desenvolvimento contínuo

Talvez você queira continuar desenvolvendo o projeto Learning Log após o envio inicial a um servidor ativo ou desenvolver seus próprios projetos para implantá-los. Há um processo razoavelmente consistente para atualizar projetos.

Inicialmente, você fará qualquer alteração necessária em seu projeto local. Se suas alterações resultarem em algum arquivo novo, acrescente esses arquivos no repositório do Git usando o comando `git add .` (lembre-se de incluir o ponto no final do comando). Qualquer mudança que exija uma migração de banco de dados exigirá esse comando, pois cada migração gera um novo arquivo de migração.

Em seguida faça commit das alterações em seu repositório usando `git`

`commit -am "`*`mensagem de commit`*`"`. Depois disso, envie suas alterações para o Heroku usando o comando `git push heroku master`. Se você migrou seu banco de dados localmente, será necessário migrar o banco de dados ativo também. Você pode usar o comando `heroku run python manage.py migrate` uma única vez ou abrir uma sessão remota de terminal com `heroku run bash` e executar o comando `python manage.py migrate`. Então acesse o seu projeto ativo e garanta que as alterações que você espera ver tenham tido efeito.

É fácil cometer erros durante esse processo, portanto não fique surpreso se algo der errado. Se o código não funcionar, revise o que você fez e tente identificar o erro. Se você não conseguir encontrá-lo ou não descobrir como corrigi-lo, veja as sugestões para obter ajuda no Apêndice C. Não tenha vergonha de pedir ajuda: todos aprenderam a construir projetos fazendo as mesmas perguntas que você provavelmente fará, de modo que alguém ficará feliz em ajudá-lo. Resolver cada problema que surgir ajudará você a desenvolver suas habilidades de forma consistente até que esteja construindo projetos significativos e confiáveis; assim você responderá às perguntas de outras pessoas também.

Configuração SECRET_KEY

O Django utiliza o valor da configuração `SECRET_KEY` em *settings.py* para implementar diversos protocolos de segurança. Neste projeto, fizemos o commit de nosso arquivo de configuração no repositório com o parâmetro `SECRET_KEY` incluído. Isso não é um problema em um projeto que serve como exercício, mas a configuração `SECRET_KEY` deve ser tratada com mais cuidado em um site de produção. Se você desenvolver um projeto que tenha uso significativo, não se esqueça de pesquisar a forma de tratar sua configuração `SECRET_KEY` de modo mais seguro.

Apagando um projeto no Heroku

Executar o processo de implantação diversas vezes com o mesmo projeto ou com uma série de pequenos projetos para dominar o processo de implantação é uma ótima prática. Porém será necessário saber como apagar um projeto implantado. O Heroku também pode limitar o número

de projetos que você pode hospedar gratuitamente, e você não vai querer encher sua conta com projetos usados para exercício.

Faça login no site do Heroku (*https://heroku.com/*); você será redirecionado para uma página que mostra uma lista de seus projetos. Clique no projeto que você quer apagar e você verá uma nova página com informações sobre o projeto. Clique no link **Settings** (Configurações) e faça rolagens para baixo até ver um link para apagar o projeto. Essa ação não pode ser revertida, portanto o Heroku pedirá que você confirme a solicitação de remoção do projeto fornecendo manualmente o nome desse projeto.

Se preferir trabalhar a partir de um terminal, um projeto também poderá ser apagado por meio do comando `destroy`:

```
(ll_env)learning_log$ heroku apps:destroy --app nomeapp
```

Nesse exemplo, `nomeaap` é o nome de seu projeto, que será algo como `afternoon-meadow-2775` ou `learning-log` se você o renomeou. Você será solicitado a fornecer o nome do projeto novamente para confirmar a remoção.

NOTA Apagar um projeto no Heroku não faz nada com a sua versão local do projeto. Se ninguém usou o projeto que você implantou e você está apenas exercitando o processo de implantação, é perfeitamente razoável apagar o seu projeto no Heroku e implantá-lo novamente.

FAÇA VOCÊ MESMO

20.3 – Blog ativo: Faça a implantação do projeto Blog com o qual você trabalhou no Heroku. Não se esqueça de definir `DEBUG` com `False` e alterar a configuração de `ALLOWED_HOSTS` para que sua implantação seja razoavelmente segura.

20.4 – Mais 404s: A função `get_object_or_404()` também deve ser usada nas views `new_entry()` e `edit_entry()`. Faça essa alteração, teste-a fornecendo um URL como *http://localhost:8000/new_entry/99999/* e verifique se você vê um erro 404.

20.5 – Learning Log estendido: Adicione uma funcionalidade em Learning Log e envie a alteração para a sua implantação ativa. Experimente fazer uma mudança simples, por exemplo, escrever mais sobre o projeto na página inicial. Então experimente adicionar uma funcionalidade mais sofisticada, como oferecer aos

usuários a opção de criar um assunto público. Isso exigirá um atributo chamado `public` como parte do modelo `Topic` (deverá ser definido com `False` por padrão) e um elemento de formulário na página `new_topic` que permita ao usuário mudar um assunto de privado para público. Então você precisará migrar o projeto e revisar *views.py* para que qualquer assunto público seja visível ao usuários não autenticados também. Lembre-se de migrar o banco de dados ativo depois que enviar suas alterações ao Heroku.

Resumo

Neste capítulo aprendemos a dar uma aparência simples, porém profissional, aos seus projetos usando a biblioteca Bootstrap e a aplicação django-bootstrap3. O uso do Bootstrap implica que os estilos que você escolher funcionarão de modo consistente em praticamente qualquer dispositivo utilizado pelas pessoas para acessar o seu projeto.

Conhecemos os templates do Bootstrap e usamos o template *Static top navbar* para dar uma aparência simples ao projeto Learning Log. Aprendemos a usar um jumbotron para fazer a mensagem na página inicial se destacar e vimos como estilizar todas as páginas de um site de forma consistente.

Na última parte do projeto, aprendemos a implantar um projeto nos servidores do Heroku para que qualquer pessoa possa acessá-lo. Criamos uma conta no Heroku e instalamos algumas ferramentas que ajudam a administrar o processo de implantação. Usamos o Git para fazer commit do projeto em funcionamento em um repositório e, em seguida, enviamos o repositório aos servidores do Heroku. Por fim, aprendemos a garantir a segurança de sua aplicação definindo `DEBUG=False` no servidor ativo.

Agora que concluímos o projeto Learning Log, você pode começar a desenvolver os seus próprios projetos. Comece de modo simples e garanta que o projeto funcione antes de aumentar a sua complexidade. Aproveite o seu aprendizado e boa sorte com seus projetos!

POSFÁCIO

Parabéns! Você aprendeu o básico sobre Python e aplicou o seu conhecimento em projetos significativos. Criou um jogo, visualizou alguns dados e desenvolveu uma aplicação web. A partir de agora, você poderá seguir em várias direções para continuar a desenvolver suas habilidades de programação.

Inicialmente, você deve continuar a trabalhar em projetos significativos que sejam de seu interesse. A programação é mais atraente quando resolvemos problemas relevantes e significativos, e agora você tem as habilidades para se envolver em diversos projetos. Você pode inventar seu próprio jogo ou escrever sua versão de um jogo clássico de arcade. Talvez queira explorar alguns dados que sejam importantes para você e criar visualizações que mostrem padrões e associações interessantes. Poderia criar sua própria aplicação web ou tentar emular uma de suas aplicações favoritas.

Sempre que for possível, convide outras pessoas a experimentar seus programas. Se escrever um jogo, deixa que outras pessoas o joguem. Se criar uma visualização, mostre-a para outras pessoas e veja se essa visualização faz sentido para elas. Se criar uma aplicação web, faça a sua implantação online e convide outros para experimentá-la. Ouça seus usuários e procure incorporar seu feedback em seus projetos; você se tornará um programador melhor se fizer isso.

Quando trabalhar em seus próprios projetos, você vai se deparar com problemas desafiadores ou até mesmo impossíveis de resolver sozinho. Continue achando maneiras de pedir ajuda e encontre o seu próprio lugar na comunidade Python. Junte-se ao Python User Group (Grupo de Usuários Python) local ou explore algumas comunidades Python online.

Considere também participar de uma PyCon próxima a você.

Você deve se esforçar para manter um equilíbrio entre trabalhar em projetos que sejam de seu interesse e desenvolver suas habilidades com Python em geral. Muitas fontes para aprendizado de Python estão disponíveis online e um grande número de livros sobre Python tem programadores de nível intermediário como alvo. Agora que você conhece o básico e sabe como aplicar suas habilidades vários desses recursos serão acessíveis. Trabalhar com os tutoriais e livros sobre Python expandirá diretamente o que você aprendeu aqui e aprofundará o seu conhecimento sobre programação em geral, e sobre Python em particular. Então, quando voltar a trabalhar em projetos depois de ter se concentrado em conhecer Python, você será capaz de resolver uma grande variedade de problemas de modo mais eficiente.

Parabéns por ter chegado até aqui e boa sorte em seu aprendizado contínuo!

A

INSTALANDO PYTHON

Python tem várias versões e muitas maneiras de ser configurado em cada sistema operacional. Este apêndice será útil caso a abordagem do Capítulo 1 não tenha funcionado, ou se você quiser instalar uma versão diferente de Python daquela que veio com o seu sistema.

Python no Linux

Python está incluído por padrão em quase todos os sistemas Linux, mas talvez você queira usar uma versão diferente da versão padrão. Em caso afirmativo, descubra qual é a versão de Python que você já tem instalada.

Descobrindo a versão instalada

Abra uma janela de terminal e execute o seguinte comando:

```
$ python --version
Python 2.7.6
```

O resultado mostra que a versão default é 2.7.6. No entanto, você pode ter também uma versão de Python 3 instalada. Para verificar, execute o comando a seguir:

```
$ python3 --version
Python 3.5.0
```

Python 3.5.0 também está instalado. Vale a pena executar os dois comandos antes de tentar instalar uma nova versão.

Instalando Python 3 no Linux

Se você não tem Python 3, ou se quiser instalar uma versão mais recente de Python 3, poderá fazê-lo com apenas algumas linhas. Usaremos um pacote chamado `deadsnakes`, que facilita instalar várias versões de Python:

```
$ sudo add-apt-repository ppa:fkrull/deadsnakes
```

```
$ sudo apt-get update
$ sudo apt-get install python3.5
```

Esses comandos instalarão Python 3.5 em seu sistema. O código a seguir iniciará uma sessão de terminal executando Python 3.5:

```
$ python3.5
>>>
```

Você também poderá usar esse comando quando configurar o seu editor de texto para usar Python 3 e executar programas a partir do terminal.

Python no OS X

Python já está instalado na maioria dos sistemas com OS X, mas talvez você queira usar uma versão diferente da versão default. Em caso afirmativo, descubra qual versão de Python você já tem instalada.

Descobrindo a versão instalada

Abra uma janela de terminal e digite o seguinte comando:

```
$ python --version
Python 2.7.6
```

Tente executar também o comando **`python3 --version`**. Provavelmente você obterá uma mensagem de erro, mas vale a pena verificar se a versão que você quer já está instalada.

Usando o Homebrew para instalar Python 3

Se você tem apenas Python 2 instalado, ou se tiver uma versão mais antiga de Python 3, poderá instalar a versão mais recente de Python 3 usando um pacote chamado Homebrew.

Instalando o Homebrew

O Homebrew depende do pacote Xcode da Apple, portanto abra um terminal e execute este comando:

```
$ xcode-select --install
```

Clique nos diálogos de confirmação que surgirem (isso poderá demorar um pouco, conforme a velocidade de sua conexão). Em seguida, instale o Homebrew:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Você poderá ver esse comando na página inicial do site do Homebrew em *http://brew.sh/*. Lembre-se de incluir um espaço entre `curl -fsSL` e o URL.

NOTA O `-e` nesse comando diz ao Ruby (a linguagem de programação em que o Homebrew está escrito) para executar o código baixado aqui. Execute comandos como esse somente de fontes em que você confie.

Para garantir que o Homebrew foi instalado corretamente, execute este comando:

```
$ brew doctor
Your system is ready to brew.
```

Essa saída quer dizer que você está pronto para instalar pacotes Python usando o Homebrew.

Instalando Python 3

Para instalar a versão mais recente de Python 3, execute o comando a seguir:

```
$ brew install python3
```

Vamos ver qual versão foi instalada usando o seguinte comando:

```
$ python3 --version
Python 3.5.0
$
```

Agora você pode iniciar uma sessão de terminal com Python 3 usando o comando `python3` e poderá usar esse comando para configurar o seu editor de texto para que ele execute programas Python com Python 3, e não com Python 2.

Python no Windows

Em geral, Python não está incluído por padrão no Windows, mas vale a pena verificar se ele está presente no sistema. Abra uma janela de terminal clicando com o botão direito do mouse em seu desktop ao mesmo tempo que mantém a tecla SHIFT pressionada, e então selecione **Open Command Window Here** (Abrir janela de comando aqui). Você também pode digitar **`command`** no Menu Iniciar. Na janela de terminal que aparecer, execute o comando a seguir:

```
> python --version
Python 3.5.0
```

Se vir uma saída como essa, é sinal de que Python já está instalado, mas talvez você queira instalar uma versão mais recente. Se vir uma mensagem de erro, será necessário fazer download de Python e instalá-lo.

Instalando Python 3 no Windows

Acesse *http://python.org/downloads/* e clique na versão de Python que você deseja instalar. Faça o download do instalador e, quando executá-lo, lembre-se de marcar a opção *Add Python to PATH* (Adicionar Python ao PATH). Isso permitirá usar o comando `python` em vez de fornecer o path completo do sistema para esse comando, e você não precisará modificar as variáveis de ambiente de seu sistema manualmente. Depois de instalar Python, execute o comando **`python --version`** em uma nova janela de terminal. Se funcionar, você terá concluído a instalação.

Encontrando o interpretador Python

Se o comando `python` simples não funcionar, será necessário dizer ao Windows em que lugar ele poderá encontrar o interpretador Python. Para descobrir isso, abra o seu drive C e encontre a pasta cujo nome começa com *Python* (talvez seja preciso fornecer a palavra `python` na barra de pesquisa do Windows Explorer para encontrar a pasta correta). Abra a pasta e procure um arquivo com o nome *python* em letras minúsculas. Clique com o botão direito do mouse nesse arquivo e selecione **Properties** (Propriedades); você verá o path desse arquivo no título *Location* (Localização).

Na janela do terminal, utilize o path para confirmar a versão que você acabou de instalar:

```
$ C:\\Python35\python --version
Python 3.5.0
```

Adicionando Python à sua variável de path

É irritante digitar o path completo sempre que você quiser iniciar um terminal Python, portanto adicionaremos o path ao sistema para que você possa simplesmente usar o comando `python`. Se você marcou a caixa *Add Python to PATH* (Adicionar Python ao PATH) na instalação, poderá ignorar este passo. Abra o **Control Panel** (Painel de Controle) de seu sistema, selecione **System and Security** (Sistema e Segurança) e então **System** (Sistema). Clique em **Advanced System Settings** (Configurações avançadas do sistema). Na janela apresentada, clique em **Environment Variables** (Variáveis de ambiente).

Na caixa cujo rótulo é *System variables* (Variáveis do sistema), procure uma variável chamada `Path`. Clique em **Edit** (Editar). Na caixa que aparecer, clique em *Variable value* (Valor da variável) e utilize a seta para a direita para avançar até o final da linha. Tome cuidado para não sobrescrever a variável existente; se fizer isso, clique em Cancel (Cancelar) e tente novamente. Acrescente um ponto e vírgula e o path de seu arquivo *python.exe* à variável existente:

```
%SystemRoot%\system32\...\System32\WindowsPowerShell\v1.0\;C:\Python34
```

Feche sua janela de terminal e abra outra. Isso carregará a nova variável `Path` em sua sessão de terminal. Agora, quando fornecer `python --version`, você deverá ver a versão de Python que você acabou de definir em sua variável `Path`. Você pode iniciar uma sessão de terminal Python simplesmente executando `python` em um prompt de comandos.

Palavras reservadas e funções embutidas de Python

Python tem seu próprio conjunto de palavras reservadas e funções embutidas. É importante ter conhecimento delas quando nomear variáveis.

Um desafio em programação é criar bons nomes de variáveis, que possam ser razoavelmente concisos e descritivos. Porém, você não pode usar nenhuma das palavras reservadas de Python, e não deve usar nomes de nenhuma função embutida, pois irá sobrescrever essas funções.

Nesta seção listaremos as palavras reservadas e os nomes das funções embutidas de Python para que você saiba quais nomes deverão ser evitados.

Palavras reservadas de Python

Cada uma das palavras reservadas a seguir tem um significado específico e você verá um erro se tentar utilizá-las como nomes de variáveis.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Funções embutidas de Python

Você não obterá um erro se usar uma das funções embutidas prontamente disponíveis a seguir como um nome de variável, porém sobrescreverá o comportamento dessa função:

```
abs()           divmod()     input()        open()       staticmethod()
all()           enumerate()  int()          ord()        str()
any()           eval()       isinstance()   pow()        sum()
basestring()    execfile()   issubclass()   print()      super()
bin()           file()       iter()         property()   tuple()
bool()          filter()     len()          range()      type()
bytearray()     float()      list()         raw_input()  unichr()
callable()      format()     locals()       reduce()     unicode()
chr()           frozenset()  long()         reload()     vars()
classmethod()   getattr()    map()          repr()       xrange()
cmp()           globals()    max()          reversed()   zip()
compile()       hasattr()    memoryview()   round()      __import__()
complex()       hash()       min()          set()        apply()
```

```
delattr()      help()     next()        setattr()   buffer()
dict()         hex()      object()      slice()     coerce()
dir()          id()       oct()         sorted()    intern()
```

NOTA Em Python 2.7, `print` é uma palavra reservada, e não uma função. Além disso, `unicode()` não está disponível em Python 3. Nenhuma dessas palavras deve ser usada como nome de variável.

B

EDITORES DE TEXTO

Os programadores passam bastante tempo escrevendo, lendo e editando código, e usar um editor de texto que deixe essas tarefas o mais eficiente possível é essencial. Um editor eficiente deve dar destaque à estrutura de seu código para que você possa identificar bugs comuns enquanto estiver trabalhando. Também deve incluir indentação automática, marcadores que mostrem o tamanho apropriado da linha e atalhos de teclado para operações comuns.

Como um novo programador, você deve usar um editor que possua esses recursos, mas que não tenha uma curva de aprendizado longa. Também é conveniente conhecer um pouco dos editores mais sofisticados para que você saiba quando deve considerar fazer um upgrading.

Daremos uma olhada em um editor de qualidade para cada um dos principais sistemas operacionais: o Geany para iniciantes que trabalhem com Linux ou Windows e o Sublime Text para OS X (embora ele também funcione bem no Linux e no Windows). Também veremos o IDLE, que é o editor que acompanha o Python por padrão. Por fim, daremos uma olhada no Emacs e no vim – dois editores sofisticados, que você verá mencionados com frequência à medida que passar mais tempo programando. Usaremos *hello_world.py* como exemplo de programa para executar em cada editor.

Geany

O Geany é um editor de texto simples que permite executar quase todos os seus programas diretamente do editor. Ele também exibe a sua saída em uma janela de terminal, o que ajudará você a se sentir à vontade usando terminais.

Instalando o Geany no Linux

Você pode instalar o Geany com uma linha na maioria dos sistemas Linux:

```
$ sudo apt-get install geany
```

Se você tiver várias versões de Python instaladas, será necessário configurar o Geany para que ele utilize a versão correta. Abra o Geany, selecione **File ▸ Save As** (Arquivo ▸ Salvar como) e salve o arquivo vazio como *hello_world.py*. Digite a linha a seguir na janela de edição:

```
print("Hello Python world!")
```

Acesse **Build ▸ Set Build Commands** (Construir ▸ Definir comandos de construção). Você deverá ver os campos Compile e Execute com um comando ao lado de cada um. O Geany pressupõe que o comando correto para cada um é `python`, mas se seu sistema utiliza o comando `python3`, será necessário alterar essa informação. Em **Compile**, digite:

```
python3 -m py_compile "%f"
```

Certifique-se de que o espaçamento e o uso de letras minúsculas em seu comando Compile estejam exatamente iguais ao que foi mostrado aqui.

Utilize o comando **Execute** a seguir:

```
python3 "%f"
```

Novamente, certifique-se de que o espaçamento e o uso de letras minúsculas estejam exatamente iguais ao que foi mostrado aqui.

Instalando o Geany no Windows

Você pode fazer o download de um instalador Windows para o Geany acessando *http://www.geany.org/* e clicando em Releases (Versões) no menu Download. Execute o instalador chamado *Geany-1.25_setup.exe* ou algo semelhante e aceite todos os defaults.

Abra o Geany, selecione **File ▸ Save As** (Arquivo ▸ Salvar como) e salve o arquivo vazio como *hello_world.py*. Digite a linha a seguir na janela de edição:

```
print("Hello Python world!")
```

Agora acesse **Build ▸ Set Build Commands** (Construir ▸ Definir comandos de construção). Você deverá ver os campos Compile e Execute com um comando ao lado de cada um. Cada um desses comandos começa com `python` (em letras minúsculas), mas o Geany não sabe em que lugar o seu sistema armazenou o comando `python`. É preciso acrescentar o path usado ao iniciar uma sessão de terminal. (Você poderá ignorar esses passos se a variável `Path` foi configurada conforme descrito no Apêndice A.)

Nos comandos Compile e Execute, acrescente o drive em que está o seu comando `python` e a pasta em que esse comando está armazenado. Seu comando **Compile** deve ter o seguinte aspecto:

```
C:\Python35\python -m py_compile "%f"
```

Seu path pode ser um pouco diferente, mas certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas estejam exatamente de acordo com o que foi mostrado aqui.

Seu comando **Execute** deve ser semelhante a:

```
C:\Python35\python "%f"
```

Novamente, certifique-se de que o espaçamento e o uso de letras maiúsculas e minúsculas em seu comando Execute estejam exatamente iguais ao que foi mostrado aqui. Quando essas linhas estiverem definidas corretamente, clique em **OK**. Agora você deverá ser capaz de executar seu programa com sucesso.

Executando programas Python no Geany

Há três maneiras de executar um programa no Geany. Para executar *hello_world.py*, selecione **Build ▸ Execute** (Construir ▸ Executar) no menu, clique no ícone com um conjunto de engrenagens ou pressione F5. Ao executar *hello_world.py*, você deverá ver uma janela de terminal com a saída a seguir:

```
Hello Python world!

------------------
(program exited with code: 0)
```

```
Press return to continue
```

Personalizando as configurações do Geany

Agora vamos configurar o Geany para que seja o mais eficiente possível personalizando os recursos mencionados no início deste apêndice.

Convertendo tabulações em espaços

Misturar tabulações com espaços em seu código pode causar problemas muito difíceis de diagnosticar em seus programas Python. Para verificar as configurações de indentação no Geany, acesse **Edit ▸ Preferences ▸ Editor ▸ Indentation** (Editar ▸ Preferências ▸ Editor ▸ Indentação). Defina o tamanho da tabulação com **4** e **Type** (Tipo) com **Spaces** (Espaços).

Se você tiver uma mistura de tabulações e espaços em um de seus programas, poderá converter todas as tabulações em espaços usando **Document ▸ Replace Tabs by Spaces** (Documento ▸ Substituir tabs por espaços).

Configurando o indicador de tamanho de linha

A maioria dos editores permite configurar uma indicação visual – em geral, é uma linha vertical – para mostrar em que ponto suas linhas devem terminar. Configure esse recurso selecionando **Edit ▸ Preferences ▸ Editor ▸ Display** (Editar ▸ Preferências ▸ Editor ▸ Exibição) e garanta que **Long line marker** (Marcador de linha longa) esteja habilitado. Em seguida, garanta que o valor de Column (Coluna) esteja definido com 79.

Indentando e diminuindo a indentação de blocos de código

Para indentar um bloco de código, marque o código e acesse **Edit ▸ Format ▸ Increase Indent** (Editar ▸ Formatar ▸ Aumentar indentação) ou pressione CTRL-I. Para diminuir a indentação de um bloco de código, acesse **Edit ▸ Format ▸ Decrease Indent** (Editar ▸ Formatar ▸ Diminuir indentação) ou pressione CTRL-U.

Comentando blocos de código

Para desabilitar temporariamente um bloco de código, você pode marcar esse bloco e comentá-lo para que Python o ignore. Acesse **Edit ▸ Format ▸ Toggle Line Commentation** (Editar ▸ Formatar ▸ Alternar comentário de linha) ou pressione CTRL-E. A linha será comentada com uma sequência especial (`#~`) para indicar que não é um comentário comum. Quando quiser remover o comentário do bloco de código, marque-o e execute o mesmo comando novamente.

Sublime Text

O Sublime Text é um editor de texto simples, fácil de instalar no OS X (e em outros sistemas também), e que permite executar quase todos os programas diretamente do editor. Ele também executa o seu código em uma sessão de terminal incluída na janela do Sublime Text, o que facilita ver a saída.

O Sublime Text tem uma política de licença bem liberal: você pode usar o editor gratuitamente durante o tempo que quiser, mas o autor pede que você compre uma licença se gostar dele e quiser continuar usando-o. Faremos o download do Sublime Text 3, que era a versão mais recente na ocasião em que escrevemos este livro.

Instalando o Sublime Text no OS X

Faça o download do instalador do Sublime Text a partir de *http://www.sublimetext.com/3*. Siga o link para download e clique no instalador para OS X. Após ter baixado o instalador, abra-o e arraste o ícone do Sublime Text para a sua pasta *Applications*.

Instalando o Sublime Text no Linux

Na maioria dos sistemas Linux, é mais fácil instalar o Sublime Text a partir de uma sessão de terminal, assim:

```
$ sudo add-apt-repository ppa:webupd8team/sublime-text-3
$ sudo apt-get update
$ sudo apt-get install sublime-text-installer
```

Instalando o Sublime Text no Windows

Faça o download de um instalador para Windows a partir de *http://www.sublimetext.com/3*. Execute o instalador; você deverá ver o Sublime Text em seu menu Iniciar.

Executando programas Python no Sublime Text

Se você usa a versão de Python que veio com o seu sistema, provavelmente poderá executar seus programas sem ajustar nenhuma configuração. Para executar programas, acesse **Tools ▸ Build** (Ferramentas ▸ Construir) ou tecle CTRL-B. Quando executar *hello_world.py*, você deverá ver uma tela de terminal na parte inferior da janela do Sublime Text mostrando a saída a seguir:

```
Hello Python world!
[Finished in 0.1s]
```

Configurando o Sublime Text

Se você tiver várias versões de Python instaladas ou se o Sublime Text não executar os programas Python automaticamente, será necessário criar um arquivo de configuração. Em primeiro lugar, você deverá saber qual é o path completo de seu interpretador Python. No Linux e no OS X, execute o seguinte comando:

```
$ type -a python3
python3 is /usr/local/bin/python3
```

Substitua `python3` pelo comando que você normalmente utiliza para iniciar uma sessão de terminal.

Se você usa Windows, consulte a seção "Instalando Python 3 no Windows", para descobrir qual é o path de seu interpretador Python.

Agora abra o Sublime Text e acesse **Tools ▸ Build System ▸ New Build System** (Ferramentas ▸ Sistema de construção ▸ Novo sistema de construção), que abrirá um novo arquivo de configuração para você. Apague o que vir e insira o seguinte:

Python3.sublime-build

```
{
    "cmd": ["/usr/local/bin/python3", "-u", "$file"],
}
```

Esse código diz ao Sublime Text para utilizar o comando `python3` quando executar o arquivo aberto no momento. Garanta que o path encontrado no passo anterior (no Windows, seu path será algo como *C:/Python35/python*) seja usado. Salve o arquivo como *Python3.sublime-build* no diretório default aberto pelo Sublime Text quando você selecionar Save (Salvar).

Abra *hello_world.py*, selecione **Tools ▸ Build System ▸ Python3** (Ferramentas ▸ Sistema de construção ▸ Python3) e, depois, **Tools ▸ Build** (Ferramentas ▸ Construir). Você deverá ver a sua saída em um terminal embutido na parte inferior da janela do Sublime Text.

Personalizando as configurações do Sublime Text

Agora vamos configurar o Sublime Text para que seja o mais eficiente possível personalizando os recursos mencionados no início deste apêndice.

Convertendo tabulações em espaços

Acesse **View ▸ Indentation** (Visualizar ▸ Indentação) e garanta que Indent Using Spaces (Indentar usando espaços) esteja marcado. Se não estiver, marque-o.

Configurando o indicador de tamanho de linha

Acesse **View ▸ Ruler** (Visualizar ▸ Régua) e clique em **80**. O Sublime Text colocará uma linha vertical na marca do caractere de número 80.

Indentando e diminuindo a indentação de blocos de código

Para indentar um bloco de código, marque-o e selecione **Edit ▸ Line ▸ Indent** (Editar ▸ Linha ▸ Indentar) ou tecle CTRL-]. Para diminuir a indentação de um bloco de código, clique em **Edit ▸ Line ▸ Unindent** (Editar ▸ Linha ▸ Remover indentação) ou tecle CTRL-[.

Comentando blocos de código

Para comentar um bloco de código marcado, selecione **Edit ▸ Comment ▸ Toggle Comment** (Editar ▸ Comentário ▸ Alternar comentário) ou tecle CTRL-/. Para remover o comentário de um bloco de código, execute o mesmo comando novamente.

IDLE

O IDLE é o editor-padrão de Python. Ele é um pouco menos intuitivo para trabalhar em comparação com o Geany e o Sublime Text, mas você verá referências a ele em outros tutoriais voltados para iniciantes; desse modo, talvez você queira experimentá-lo.

Instalando o IDLE no Linux

Se você usa Python 3, instale o pacote `idle3`, assim:

```
$ sudo apt-get install idle3
```

Se você usa Python 2, instale o pacote `idle` desta maneira:

```
$ sudo apt-get install idle
```

Instalando o IDLE no OS X

Se você usou o Homebrew para instalar Python, o IDLE provavelmente já estará em seu sistema. Em um terminal, execute o comando **`brew linkapps`**, que diz ao IDLE como localizar o interpretador Python correto em seu sistema. Então você encontrará o IDLE em sua pasta de aplicativos do usuário.

Do contrário, acesse *https://www.python.org/download/mac/tcltk/* e siga as instruções que estão aí; você também precisará instalar alguns pacotes gráficos dos quais o IDLE depende.

Instalando o IDLE no Windows

O IDLE deve ter sido instalado automaticamente quando você instalou Python. Ele deverá estar em seu menu Iniciar.

Personalizando as configurações do IDLE

Como é o editor-padrão de Python, a maioria das configurações do IDLE já estará ajustada conforme os parâmetros recomendados para Python: as tabulações serão automaticamente convertidas em espaços e o indicador de tamanho de linha estará configurado para 80 caracteres.

Indentando e diminuindo a indentação de blocos de código

Para indentar um bloco de código, marque-o e selecione **Format ▸ Indent Region** (Formatar ▸ Indentar região) ou tecle CTRL-]. Para diminuir a indentação de um bloco de código, selecione **Format ▸ Dedent Region** (Formatar ▸ Diminuir a indentação da região) ou tecle CTRL-[.

Comentando blocos de código

Para comentar um bloco de código, marque-o e selecione **Format ▸ Comment Out Region** (Formatar ▸ Comentar região) ou tecle ALT-3. Para remover o comentário do código, selecione **Format ▸ Uncomment Region** (Formatar ▸ Remover comentário da região) ou tecle ALT-4.

Emacs e o vim

O Emacs e o vim são dois editores populares preferidos por muitos programadores experientes, pois foram projetados para serem usados de modo que suas mãos jamais precisem deixar o teclado. Isso faz com que escrever, ler e modificar código sejam operações bem eficientes, depois que você souber como o editor funciona. Também quer dizer que eles têm uma curva de aprendizado razoavelmente longa.

Os programadores, com frequência, recomendarão que você os experimente, mas muitos programadores proficientes se esquecem da grande quantidade de informações que os novos programadores já estão tentando aprender. É interessante saber da existência desses editores, mas evite usá-los até se sentir confortável em escrever e em trabalhar com códigos em um editor mais simples, que lhe permita se concentrar em aprender a programar, e não em aprender a usar um editor.

C

OBTENDO AJUDA

Todos ficam sem saber o que fazer em algum momento quando estão aprendendo a programar, e uma das habilidades mais importantes para adquirir como programador é como sair dessa situação de modo eficiente. Este apêndice apresenta várias maneiras de ajudar você a sair de uma situação confusa em programação.

Primeiros passos

Quando não souber o que fazer, seu primeiro passo deverá ser avaliar a sua situação. Antes de poder obter ajuda de outras pessoas, você deverá ser capaz de responder a estas três perguntas claramente:

- O que você está tentando fazer?
- O que você já tentou fazer até agora?
- Quais resultados você está obtendo?

Suas respostas devem ser as mais específicas possível. Para a primeira pergunta, declarações explícitas como "Estou tentando instalar a versão mais recente de Python 3 em meu computador com Windows 10" são detalhadas o suficiente para que outras pessoas da comunidade Python possam ajudar você. Frases como "Estou tentando instalar Python" não oferecem informações suficientes para que outras pessoas possam ajudar.

Sua resposta à segunda pergunta deve oferecer detalhes suficientes a ponto de você não ser aconselhado a repetir o que já tentou fazer: "Acessei *http://python.org/downloads/* e cliquei no botão Download para Python 3. Então executei o instalador" é mais útil que "Acessei o site de Python e fiz download de um instalador".

Para a última pergunta, é útil saber as mensagens de erro exatas obtidas quando estiver pesquisando online em busca de uma solução ou quando pedir ajuda.

Às vezes, responder a essas três perguntas para si mesmo permite ver algo que tenha passado despercebido e resolver o problema sem precisar de outros recursos. Os programadores têm até mesmo um nome para isso: chama-se *rubber duck debugging* (depuração com patinho de borracha). Se você explicar sua situação a um patinho de borracha (ou qualquer objeto inanimado) de forma clara e lhe fizer uma pergunta específica, muitas vezes você será capaz de responder à sua própria pergunta. Algumas empresas de programação até mesmo mantêm um patinho de borracha de verdade por aí para incentivar as pessoas a "conversar com o patinho".

Tente novamente

Simplesmente voltar ao início e tentar novamente pode ser suficiente para resolver muitos problemas. Suponha que você esteja tentando escrever um laço `for` baseado em um exemplo deste livro. Você pode ter se esquecido de algo simples, por exemplo, colocar dois-pontos no final da linha `for`. Executar os passos novamente pode ajudar a evitar que você repita o mesmo erro.

Faça uma pausa

Se você estiver trabalhando no mesmo problema por um tempo, fazer uma pausa, na verdade, é uma das melhores táticas que você poderá tentar. Quando trabalhamos por longos períodos de tempo na mesma tarefa, nosso cérebro passa a enfocar uma única solução. Deixamos de ver as pressuposições que fizemos, e fazer uma pausa nos ajuda a ver o problema com uma nova perspectiva. Não é preciso fazer uma pausa longa: basta ser algo que tire você do processo de raciocínio atual. Se você está sentado há um bom tempo, faça uma atividade física: dê uma caminhada rápida ou saia da sala por um instante; talvez você possa beber um copo de água ou fazer um lanche leve e saudável.

Se estiver ficando frustrado, pode ser que valha a pena deixar o seu trabalho de lado pelo resto do dia. Uma boa noite de sono quase sempre deixa o problema mais fácil de ser encarado.

Consulte os recursos deste livro

Os recursos online deste livro, disponíveis em *https://www.nostarch.com/pythoncrashcourse/*, incluem diversas seções úteis sobre configuração de seu sistema e como trabalhar com cada capítulo. Se você ainda não fez isso, dê uma olhada nesses recursos e veja se há algo que possa ajudar.

Pesquisando online

Há uma boa chance de que outra pessoa já tenha tido o mesmo problema que você e que tenha escrito sobre ele online. Boas habilidades de pesquisa e perguntas específicas ajudarão você a encontrar recursos existentes para resolver o problema que você está enfrentando. Por exemplo, se você está tendo dificuldades para instalar Python 3 no Windows 10, pesquisar os termos *python 3 windows 10* pode direcioná-lo a uma resposta.

Pesquisar a mensagem de erro exata também pode ser extremamente útil. Por exemplo, suponha que você tenha obtido o seguinte erro ao tentar iniciar uma sessão de terminal Python:

```
> python
'python' is not recognized as an internal or external command
>
```

Pesquisar a frase completa *python is not recognized as an internal or external command* (python não é reconhecido como um comando interno ou externo) provavelmente resultará em bons conselhos.

Ao começar a pesquisar assuntos relacionados à programação, alguns sites aparecerão repetidamente. Descreverei alguns desses sites de forma rápida para que você saiba como eles poderão ser úteis.

Stack Overflow

O Stack Overflow (*http://stackoverflow.com/*) é um dos sites mais populares de perguntas e respostas para programadores e, com frequência, aparecerá na primeira página de resultados de pesquisas relacionadas a Python. Os participantes postam perguntas quando não sabem o que fazer, e outros

membros tentam dar respostas úteis. Os usuários podem votar nas respostas que acharem mais úteis; desse modo, as melhores respostas geralmente são as primeiras que você verá.

Muitas perguntas básicas de Python têm respostas bem claras no Stack Overflow, pois a comunidade as têm aperfeiçoado com o passar do tempo. Os usuários são incentivados a postar atualizações também, de modo que as respostas tendem a permanecer relativamente atualizadas. Na época em que este livro foi escrito, mais de 400 mil perguntas relacionadas a Python haviam sido respondidas no Stack Overflow.

Documentação oficial de Python

A documentação oficial de Python (*http://docs.python.org/*) é como um tiro no escuro para os iniciantes, pois seu propósito é mais documentar a linguagem que redigir explicações. Os exemplos na documentação oficial devem funcionar, mas talvez você não compreenda tudo que é apresentado. Apesar disso, é um bom recurso a ser consultado quando aparecer em suas pesquisas, e se tornará mais útil à medida que você continuar a expandir sua compreensão de Python.

Documentação oficial da biblioteca

Se você usa uma biblioteca específica, por exemplo, o Pygame, o matplotlib, Django e assim por diante, links para a documentação oficial do respectivo projeto aparecerão com frequência nas pesquisas – por exemplo, *http://docs.djangoproject.com/* é bem útil. Se você planeja trabalhar com qualquer uma dessas bibliotecas, é uma boa ideia ter familiaridade com a documentação oficial.

r/learnpython

O Reddit é composto de vários subfóruns chamados de *subreddits*. O subreddit *r/learnpython* (*http://reddit.com/r/learnpython/*) é bem ativo e oferece bastante ajuda. Nesse local você pode ler perguntas de outras pessoas e postar suas próprias perguntas.

Postagens de blog

Muitos programadores mantêm blogs e compartilham postagens sobre as partes da linguagem com as quais estão trabalhando. Dê uma olhada rápida nos primeiros comentários de uma postagem de blog para ver as reações das outras pessoas antes de aceitar qualquer conselho. Se nenhum comentário surgir, considere a postagem com certa dose de reserva. É possível que ninguém mais tenha verificado o conselho.

IRC (Internet Relay Chat)

Os programadores interagem em tempo real no IRC. Se você não souber como resolver um problema e pesquisas online não estão oferecendo respostas, perguntar em um canal de IRC pode ser a sua melhor opção. A maioria das pessoas que participa desses canais é bem-educada e solícita, em especial se você puder ser específico em relação ao que está tentando fazer, o que já tentou e quais foram os resultados obtidos.

Crie uma conta no IRC

Para criar uma conta no IRC, acesse *http://webchat.freenode.net/*. Escolha um apelido, preencha a caixa Captcha e clique em **Connect** (Conectar). Você verá uma mensagem lhe dando as boas-vindas ao servidor IRC freenode. Na caixa na parte inferior da janela, digite o comando a seguir:

```
/msg nickserv register senha email
```

Forneça sua própria senha e seu endereço de email no lugar de *senha* e de *email*. Escolha uma senha simples, que você não use em nenhuma outra conta. Essa senha não é transmitida de modo seguro, portanto nem mesmo tente criar uma senha segura. Você receberá um email com instruções para verificar a sua conta. O email fornecerá um comando como este:

```
/msg nickserv verify register apelido código_de_verificação
```

Copie e cole essa linha no site do IRC usando o nome que você escolheu antes como o *apelido* e um valor para o *código_de_verificação*. Agora você estará pronto para se associar a um canal.

Canais para se associar

Para se associar ao canal principal de Python, forneça **`/join #python`** na caixa de entrada. Você verá uma confirmação de que você se associou ao canal e algumas informações gerais sobre ele.

O canal *##learnpython* (com dois sinais de sustenido) geralmente é bem ativo também. Esse canal está associado a *http://reddit.com/r/learnpython/*, portanto você verá mensagens sobre postagens em *r/learnpython* também. O canal *#pyladies* tem como foco o apoio a mulheres que estão aprendendo Python, assim como às pessoas que dão apoio às programadoras do sexo feminino. Talvez você queira se associar ao canal *#django* se estiver trabalhando com aplicações web.

Depois de se associar a um canal, você poderá ler as conversas de outras pessoas, além de fazer suas próprias perguntas.

Cultura do IRC

Para obter uma ajuda eficaz, você deve conhecer alguns detalhes da cultura do IRC. Manter o foco nas três perguntas do início deste apêndice definitivamente ajudará a direcionar você para uma solução bem-sucedida. As pessoas ficarão satisfeitas em ajudar se você puder explicar exatamente o que está tentando fazer, o que já tentou fazer e os resultados específicos que está obtendo. Se precisar compartilhar códigos ou uma saída, os membros do IRC utilizam sites externos criados para esse propósito, como o *https://bpaste.net/+python/*. (É para esse site que `#python` envia você para compartilhar códigos e saídas.) Isso evita que os canais fiquem lotados de código e também facilita bastante ler o código compartilhado pelas pessoas.

Ser paciente sempre deixará as pessoas mais dispostas a ajudar você. Faça suas perguntas de modo conciso e espere que alguém responda. Muitas vezes as pessoas estão no meio de várias conversas, mas, em geral, alguém responderá em um período de tempo razoável. Se houver poucas pessoas no canal, talvez demore um pouco para que você obtenha uma resposta.

D

USANDO O GIT PARA CONTROLE DE VERSÕES

O software de controle de versões permite criar snapshots (imagem instantânea) de um projeto sempre que ele estiver funcionando. Ao fazer alterações em um projeto – por exemplo, quando implementar uma nova funcionalidade – você tem a opção de retornar a um estado anterior em funcionamento caso o estado do projeto não esteja correto no momento.

Usar um software de controle de versões oferece a você a liberdade para trabalhar em melhorias e cometer erros sem se preocupar em arruinar o seu projeto. Isso é especialmente importante em projetos de grande porte, mas também pode ser útil em projetos menores, mesmo quando você estiver trabalhando em programas contidos em um único arquivo.

Neste apêndice aprenderemos a instalar o Git e a usá-lo para controle de versões nos programas em que você está trabalhando no momento. O Git é o software mais popular de controle de versões usado atualmente. Muitas de suas ferramentas sofisticadas ajudam as equipes a colaborar em projetos grandes, mas seus recursos mais básicos também funcionam bem para desenvolvedores que trabalhem sozinhos. O Git implementa controle de versões monitorando as alterações feitas em todos os arquivos de um projeto; se você cometer um erro, poderá simplesmente retornar a um estado salvo anteriormente.

Instalando o Git

O Git executa em todos os sistemas operacionais, porém há abordagens diferentes para instalá-lo em cada sistema. As próximas seções apresentam instruções específicas para cada sistema operacional.

Instalando o Git no Linux

Para instalar o Git no Linux, execute o seguinte comando:

```
$ sudo apt-get install git
```

É isso. Agora você pode usar o Git em seus projetos.

Instalando o Git no OS X

O Git talvez já esteja instalado em seu sistema, portanto experimente executar o comando **`git --version`**. Se vir uma saída que liste um número específico de versão, é sinal de que o Git está instalado em seu sistema. Se obtiver uma mensagem pedindo que você instale ou atualize o Git, basta seguir as instruções da tela.

Você também pode acessar *https://git-scm.com/*, seguir o link Downloads e clicar em um instalador apropriado ao seu sistema.

Instalando o Git no Windows

Você pode instalar o Git para Windows a partir de *http://msysgit.github.io/*.

Configurando o Git

O Git controla quem fez alterações em um projeto, mesmo nos casos em que há apenas uma pessoa trabalhando nele. Para isso o Git precisa ter o seu nome de usuário e o seu email. Você deve fornecer um nome de usuário, mas sinta-se à vontade para criar um endereço de email falso:

```
$ git config --global user.name "nome_do_usuário"
$ git config --global user.email "nome_do_usuário@exemplo.com"
```

Se você se esquecer desse passo, o Git solicitará essas informações quando você fizer o seu primeiro commit.

Criando um projeto

Vamos criar um projeto com o qual trabalharemos. Crie uma pasta chamada *git_practice* em algum lugar de seu sistema. Nessa pasta, crie um programa Python simples:

hello_world.py

```
print("Hello Git world!")
```

Usaremos esse programa para explorar as funcionalidades básicas do Git.

Ignorando arquivos

Arquivos com a extensão *.pyc* são gerados automaticamente a partir de arquivos *.py*, portanto não precisamos que o Git mantenha o controle sobre eles. Esses arquivos são armazenados em um diretório chamado *__pycache__*. Para dizer ao Git que ignore esse diretório, crie um arquivo especial de nome *.gitignore* – com um ponto no início do nome do arquivo e sem extensão – e acrescente a linha a seguir nesse arquivo:

.gitignore

```
__pycache__/
```

Isso diz ao Git para ignorar qualquer arquivo que estiver no diretório *__pycache__*. O uso de um arquivo *.gitignore* evita que seu projeto fique entulhado e faz com que seja mais fácil trabalhar com ele.

NOTA Se você usa Python 2.7, substitua essa linha por `*.pyc`. O Python 2.7 não cria um diretório *__pycache__*; cada arquivo *.pyc* é armazenado no mesmo diretório em que está o arquivo *.py* correspondente. O asterisco diz ao Git para ignorar qualquer arquivo com extensão *.pyc*.

Talvez seja necessário modificar as configurações de seu editor de texto para que ele mostre os arquivos ocultos a fim de abrir o arquivo *.gitignore*. Alguns editores são configurados para ignorar nomes de arquivos que comecem com um ponto.

Inicializando um repositório

Agora que você tem um diretório contendo um arquivo Python e um arquivo *.gitignore*, um repositório do Git poderá ser inicializado. Abra um terminal, navegue até a pasta *git_practice* e execute o comando a seguir:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

A saída mostra que o Git inicializou um repositório vazio em *git_practice*.

Um *repositório* é o conjunto de arquivos de um programa que o Git está monitorando de forma ativa. Todos os arquivos que o Git usa para administrar o repositório estão localizados no diretório oculto *.git/*, com o qual você não precisará trabalhar. Basta não apagar esse diretório; do contrário, você perderá o histórico de seu projeto.

Verificando o status

Antes de fazer qualquer outra tarefa, vamos ver o status do projeto:

```
  git_practice$ git status
u # On branch master
  #
  # Initial commit
  #
v # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
  #
  #   .gitignore
  #   hello_world.py
  #
w nothing added to commit but untracked files present (use "git add" to
track)
  git_practice$
```

No Git, um *branch* é uma versão do projeto em que você está trabalhando; nesse caso, podemos ver que estamos em um branch chamado `master` u. Sempre que verificar o status de seu projeto, ele deverá informar que você está no branch `master`. Em seguida, podemos ver que estamos prestes a fazer o commit inicial. Um *commit* é um snapshot do projeto em determinado ponto no tempo.

O Git nos informa que há arquivos não monitorados no projeto v porque não lhe dissemos ainda quais arquivos devem ser controlados. Em seguida, fomos informados de que nada foi adicionado no commit atual, mas existem arquivos não monitorados que talvez possamos querer acrescentar ao repositório w.

Adicionando arquivos no repositório

Vamos adicionar os dois arquivos no repositório e verificar o status novamente:

```
u git_practice$ git add .
v git_practice$ git status
  # On branch master
  #
  # Initial commit
  #
  # Changes to be committed:
  #   (use "git rm --cached <file>..." to unstage)
  #
w #    new file:   .gitignore
  #   new file:   hello_world.py
  #
  git_practice$
```

O comando `git add .` adiciona todos os arquivos de um projeto que ainda não estão sendo monitorados ao repositório u. Esse comando não faz commit dos arquivos; ele simplesmente diz ao Git para começar a prestar atenção neles. Quando verificamos o status do projeto agora, podemos ver que o Git reconhece algumas alterações que precisam de commit v. O rótulo *new file* (arquivo novo) quer dizer que esses arquivos acabaram de ser adicionados ao repositório w.

Fazendo um commit

Vamos fazer o primeiro commit:

```
u git_practice$ git commit -m "Started project."
v [master (root-commit) c03d2a3] Started project.
w 2 files changed, 1 insertion(+)
  create mode 100644 .gitignore
  create mode 100644 hello_world.py
x git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
  git_practice$
```

Executamos o comando `git commit -m "mensagem"` **u** para criar um snapshot do projeto. A flag `-m` diz ao Git para registrar a mensagem que vem a seguir (`"Started project."`) no log do projeto. A saída mostra que estamos no branch `master` **v** e que dois arquivos foram alterados **w**.

Quando verificamos o status agora, podemos ver que estamos no branch `master` e que temos um diretório de trabalho limpo **x**. Essa é a mensagem que você vai querer ver sempre que fizer um commit de um estado de seu projeto em funcionamento. Se obtiver uma mensagem diferente, leia-a com atenção; é provável que você tenha se esquecido de adicionar um arquivo antes de fazer um commit.

Verificando o log

O Git mantém um log de todos os commits feitos no projeto. Vamos verificar o log:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f
Author: Eric Matthes <eric@example.com>
Date:   Mon Mar 16 07:23:32 2015 -0800

    Started project.
git_practice$
```

Sempre que você faz um commit, o Git gera um ID de referência único com 40 caracteres. Ele registra quem fez o commit, quando ele foi feito e a mensagem registrada. Nem sempre você precisará de todas essas informações, por isso o Git oferece uma opção para exibir uma versão mais simples das entradas do log:

```
git_practice$ git log --pretty=oneline
a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f Started project.
git_practice$
```

A flag `--pretty=oneline` oferece as duas informações mais importantes: o ID de referência e a mensagem registrada no commit.

Segundo commit

Para ver a verdadeira eficácia do sistema de controle de versões, precisamos fazer uma alteração no projeto e fazer o commit dessa mudança. Nesse caso, simplesmente acrescentaremos outra linha em *hello_world.py*:

hello_world.py

```
print("Hello Git world!")
print("Hello everyone.")
```

Se conferirmos o status do projeto, veremos que o Git percebeu qual arquivo sofreu alterações:

```
  git_practice$ git status
u # On branch master
  # Changes not staged for commit:
  #   (use "git add <file>..." to update what will be committed)
  #   (use "git checkout -- <file>..." to discard changes in working
  directory)
  #
v #   modified:   hello_world.py
  #
w no changes added to commit (use "git add" and/or "git commit -a")
  git_practice$
```

Vemos o branch em que estamos trabalhando u, o nome do arquivo modificado v e que não houve commit de nenhuma alteração w. Vamos fazer commit da alteração e verificar o status novamente:

```
u git_practice$ git commit -am "Extended greeting."
  [master 08d4d5e] Extended greeting.
  1 file changed, 1 insertion(+)
v git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
w git_practice$ git log --pretty=oneline
  08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Extended greeting.
  be017b7f06d390261dbc64ff593be6803fd2e3a1 Started project.
  git_practice$
```

Fizemos um novo commit, passando a flag `-am` quando usamos o comando `git commit` u. A flag `-a` diz ao Git para adicionar todos os arquivos modificados no repositório ao commit atual. (Se você criar algum

arquivo novo entre os commits, basta executar o comando `git add .` novamente para incluir os novos arquivos no repositório.) A flag `-m` diz ao Git para registrar uma mensagem no log para esse commit.

Quando verificamos o status do projeto, vemos que, mais uma vez, temos um diretório de trabalho limpo v. Por fim, vemos os dois commits no log w.

Revertendo uma alteração

Vamos agora ver como abandonar uma alteração e reverter para o estado anterior em funcionamento. Inicialmente acrescente uma nova linha em *hello_world.py*:

hello_world.py

```
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Salve e execute esse arquivo.

Verificamos o status e vemos que o Git percebeu essa mudança:

```
git_practice$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
u #   modified:   hello_world.py
#
no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

O Git percebe que modificamos *hello_world.py* u e podemos fazer commit da alteração se quisermos. Porém, dessa vez, no lugar de fazer commit da alteração, queremos retornar para o último commit, quando sabíamos que o nosso projeto estava funcionando. Não faremos nada com

hello_world.py; não apagaremos a linha nem usaremos o recurso Undo (Desfazer) do editor de texto. Em vez disso, execute os comandos a seguir em sua sessão de terminal:

```
git_practice$ git checkout .
git_practice$ git status
# On branch master
nothing to commit, working directory clean
git_practice$
```

O comando `git checkout` permite trabalhar com qualquer commit anterior. O comando `git checkout .` abandona qualquer alteração feita desde o último commit e restaura o projeto ao estado do último commit.

Quando retornar ao seu editor de texto, você verá que *hello_world.py* voltou a ser:

```
print("Hello Git world!")
print("Hello everyone.")
```

Embora retornar a um estado anterior possa parecer trivial nesse projeto simples, se estivéssemos trabalhando em um projeto grande, com dezenas de arquivos modificados, todos os arquivos alterados desde o último commit teriam sido restaurados ao estado anterior. Esse recurso é extremamente útil: você pode fazer quantas alterações quiser ao implementar um novo recurso e, caso ele não funcione, poderá descartá-lo sem prejudicar o projeto. Não é preciso lembrar quais foram as alterações nem desfazê-las manualmente. O Git faz tudo isso para você.

NOTA Talvez você precise clicar na janela de seu editor para atualizar o arquivo e ver a versão anterior.

Check out de commits anteriores

Você pode fazer check out de qualquer commit em seu log, e não apenas do mais recente, se incluir os seis primeiros caracteres do ID de referência no lugar de um ponto. Ao fazer um check out desse tipo, você pode rever um commit anterior e poderá então retornar ao commit mais novo ou abandonar seu trabalho recente e reestabelecer o desenvolvimento a partir

do commit mais antigo:

```
git_practice$ git log --pretty=oneline
08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Extended greeting.
be017b7f06d390261dbc64ff593be6803fd2e3a1 Started project.
git_practice$ git checkout be017b
Note: checking out 'be017b'.

u You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you
may
do so (now or later) by using -b with the checkout command again.
Example:

  git checkout -b new_branch_name

HEAD is now at be017b7... Started project.
git_practice$
```

Ao fazer check out de um commit anterior, você deixará o branch master e entrará no que o Git chama de estado *detached HEAD* u. *HEAD* é o estado atual do projeto; estamos *desassociados* (detached) porque saímos de um branch nomeado (`master`, nesse caso).

Para retornar ao branch `master`, faça check out desse branch:

```
git_practice$ git checkout master
Previous HEAD position was be017b7... Started project.
Switched to branch 'master'
git_practice$
```

Esse comando leva você de volta ao branch `master`. A menos que você queira trabalhar com alguns recursos mais sofisticados do Git, é melhor não fazer nenhuma alteração em seu projeto quando tiver feito check out de um commit antigo. No entanto, se você for a única pessoa trabalhando em um projeto e quiser descartar todos os commits mais recentes e

retornar a um estado anterior, poderá reiniciar o projeto em um commit anterior. A partir do branch `master`, digite o seguinte:

```
u git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
v git_practice$ git log --pretty=oneline
  08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Extended greeting.
  be017b7f06d390261dbc64ff593be6803fd2e3a1 Started project.
w git_practice$ git reset --hard be017b
  HEAD is now at be017b7 Started project.
x git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
y git_practice$ git log --pretty=oneline
  be017b7f06d390261dbc64ff593be6803fd2e3a1 Started project.
  git_practice$
```

Inicialmente conferimos o status para garantir que estamos no branch `master` u. Quando observamos o log, vemos os dois commits v. Então executamos o comando `git reset --hard` com os seis primeiros caracteres do ID de referência do commit para o qual queremos retornar de modo permanente w. Verificamos o status novamente e vemos que estamos no branch `master`, sem que haja nada para fazer commit x. Quando observamos o log novamente, vemos que estamos no commit a partir do qual queremos começar de novo y.

Apagando o repositório

Às vezes você deixará o histórico de seu repositório confuso e não saberá como recuperá-lo. Se isso acontecer, considere pedir ajuda antes usando os métodos discutidos no Apêndice C. Se não conseguir fazer a correção e estiver trabalhando em um projeto sozinho, você poderá continuar a trabalhar com os arquivos, mas livre-se do histórico do projeto apagando o diretório *.git*. Isso não afetará o estado atual de nenhum dos arquivos, mas apagará todos os commits, portanto você não poderá fazer check out de nenhum outro estado do projeto.

Para fazer isso, abra um explorador de arquivos e apague o repositório

.git ou faça isso a partir da linha de comando. Depois disso, será necessário recomeçar com um novo repositório e passar a monitorar suas alterações novamente. A seguir, mostramos como é o processo como um todo em uma sessão de terminal:

```
u git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
v git_practice$ rm -rf .git
w git_practice$ git status
  fatal: Not a git repository (or any of the parent directories): .git
x git_practice$ git init
  Initialized empty Git repository in git_practice/.git/
y git_practice$ git status
  # On branch master
  #
  # Initial commit
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
  #
  #   .gitignore
  #   hello_world.py
  #
  nothing added to commit but untracked files present (use "git add" to
  track)
z git_practice$ git add .
  git_practice$ git commit -m "Starting over."
  [master (root-commit) 05f5e01] Starting over.
  2 files changed, 2 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 hello_world.py
{ git_practice$ git status
  # On branch master
  nothing to commit, working directory clean
  git_practice$
```

Em primeiro lugar, verificamos o status e vemos que temos um diretório de trabalho limpo **u**. Então utilizamos o comando `rm -rf .git` para apagar o diretório *.git* (`rmdir /s .git` no Windows) **v**. Quando verificamos o status

depois de apagar a pasta *.git*, somos informados de que esse não é um repositório do Git W. Todas as informações que o Git usa para monitorar um repositório estão armazenadas na pasta *.git*, portanto removê-la faz com que todo o repositório seja apagado.

Estamos então livres para usar `git init` a fim de iniciar um novo repositório X. A verificação do status mostra que estamos de volta ao estágio inicial, à espera do primeiro commit y. Adicionamos os arquivos e fizemos o primeiro commit z. A verificação do status agora mostra que estamos no novo branch `master`, sem nada para fazer commit {.

O uso do sistema de controle de versões exige um pouco de prática, mas depois que começar a usá-lo, você nunca mais vai querer trabalhar sem ele novamente.

Black Hat Python
Programação Python para
hackers e pentesters
Justin Seitz
novatec
Apresentação de Charlie Miller

senhas e injetar shellcode em uma máquina virtual.
> Estender o Burp Suite, que é uma ferramenta popular para web hacking.
> Explorar a automação do Windows COM para realizar um ataque do tipo man-in-the-browser.
> Obter dados de uma rede, principalmente de forma sub-reptícia.

Técnicas usadas por pessoas da área e desafios criativos ao longo de toda a obra mostrarão como estender os hacks e criar seus próprios exploits.

Quando se trata de segurança ofensiva, sua habilidade para criar ferramentas eficazes de forma imediata será indispensável.

Saiba como fazer isso em Black Hat Python.

JOVEM E
BEM-SUCEDIDO
Um guia para a realização
profissional e financeira
TRABALHO
DINHEIRO
ESTUDOS
novatec
Juliano Niederauer

DINHEIRO: explica como assumir o controle de suas finanças, para, então, começar a investir e multiplicar seu patrimônio. Apresenta estratégias de investimentos de acordo com o momento de vida de cada um, abordando as vantagens e desvantagens de cada tipo de investimento.

Jovem e Bem-sucedido apresenta ideias que o acompanharão a vida toda, realizando importantes mudanças no modo como você planeja estudar, trabalhar e lidar com o dinheiro.

Wireshark para profissionais de segurança

Bullock, Jessey

9788575225998

320 páginas

[Compre agora e leia]

Um guia essencial para segurança de rede e para o Wireshark – um conjunto de ferramentas repleto de recursos

O analisador de protocolos de código aberto Wireshark é uma ferramenta de uso consagrado em várias áreas, incluindo o campo da segurança. O Wireshark disponibiliza um conjunto eficaz de recursos que permite inspecionar a sua rede em um nível microscópico. Os diversos recursos e o suporte a vários protocolos fazem do Wireshark uma ferramenta de segurança de valor inestimável, mas também o tornam difícil ou intimidador para os iniciantes que queiram conhecê-lo. Wireshark para profissionais de segurança é a resposta: ele ajudará você a tirar proveito do Wireshark e de ferramentas relacionadas a ele, por exemplo, a aplicação de linha de comando TShark, de modo rápido e eficiente. O conteúdo inclui uma introdução completa ao Metasploit, que é uma ferramenta de ataque eficaz, assim como da linguagem popular de scripting Lua.

Este guia extremamente prático oferece o insight necessário para

você aplicar o resultado de seu aprendizado na vida real com sucesso. Os exemplos mostram como o Wireshark é usado em uma rede de verdade, com o ambiente virtual Docker disponibilizado; além disso, princípios básicos de rede e de segurança são explicados em detalhes para ajudar você a entender o porquê, juntamente com o como. Ao usar a distribuição Kali Linux para testes de invasão, em conjunto com o laboratório virtual e as capturas de rede disponibilizadas, você poderá acompanhar os diversos exemplos ou até mesmo começar a pôr em prática imediatamente o seu conhecimento em um ambiente de rede seguro. A experiência prática torna-se mais valiosa ainda pela ênfase em uma aplicação coesa, ajudando você a explorar vulnerabilidades e a expandir todas as funcionalidades do Wireshark, estendendo-as ou integrando-as com outras ferramentas de segurança.

Definindo Escopo em
Projetos de Software
novatec
Carlos Alberto Debastiani

Definindo Escopo em Projetos de Software

Debastiani, Carlos Alberto

9788575224960

144 páginas

Compre agora e leia

Definindo Escopo em Projetos de Software é uma obra que pretende tratar, de forma clara e direta, a definição de escopo como o fator mais influente no sucesso dos projetos de desenvolvimento de sistemas, uma vez que exerce forte impacto sobre seus custos. Abrange diversas áreas do conhecimento ligadas ao tema, abordando desde questões teóricas como a normatização e a definição das características de engenharia de software, até questões práticas como métodos para coleta de requisitos e ferramentas para desenho e projeto de soluções sistêmicas.

Utilizando uma linguagem acessível, diversas ilustrações e citações de casos vividos em sua própria experiência profissional, o autor explora, de forma abrangente, os detalhes que envolvem a definição de escopo, desde a identificação das melhores fontes de informação e dos envolvidos na tomada de decisão, até as técnicas e ferramentas usadas no levantamento de requisitos, no projeto da solução e nos testes de aplicação.

Manual do Futuro
Redator
Técnicas e causos para quem
quer se aventurar na profissão
novatec
Sérgio Calderaro

Manual do Futuro Redator

Calderaro, Sérgio

9788575224908

120 páginas

Você estuda ou está pensando em estudar Publicidade, Jornalismo ou Letras? Gosta de ler e escrever e quer dicas de quem já passou por poucas e boas em agências de publicidade, redações de jornal e editoras? Quer conhecer causos curiosos de um profissional do texto que já deu aulas em universidades do Brasil e da Europa e trabalhou como assessor de Imprensa e Divulgação em uma das maiores embaixadas brasileiras do exterior?

O Manual do futuro redator traz tudo isso e muito mais. Em linguagem ágil e envolvente, mescla orientações técnicas a saborosas histórias do dia a dia de um profissional com duas décadas e meia de ofício. Esta obra inédita em sua abordagem pretende fazer com que você saiba onde está se metendo antes de decidir seu caminho. Daí pra frente, a decisão será sua. Vai encarar?

Compre agora e leia